US005615375A

United States Patent [19]

Ibusuki et al.

[11] Patent Number: 5,615,375

[45] Date of Patent: Mar. 25, 1997

[54] **INTERRUPT CONTROL CIRCUIT**

[75] Inventors: **Takeshi Ibusuki**; **Shinji Miyahara**, both of Kawasaki, Japan

[73] Assignee: **Fujitsu Limited**, Kawasaki, Japan

[21] Appl. No.: **153,159**

[22] Filed: **Nov. 17, 1993**

[30] **Foreign Application Priority Data**

Nov. 17, 1992 [JP] Japan 4-305753

[51] **Int. Cl.**[6] **G06F 9/46**

[52] **U.S. Cl.** **395/737**; 395/591

[58] **Field of Search** 395/725, 325, 395/775, 375, 733–742, 293, 249, 299

[56] **References Cited**

U.S. PATENT DOCUMENTS

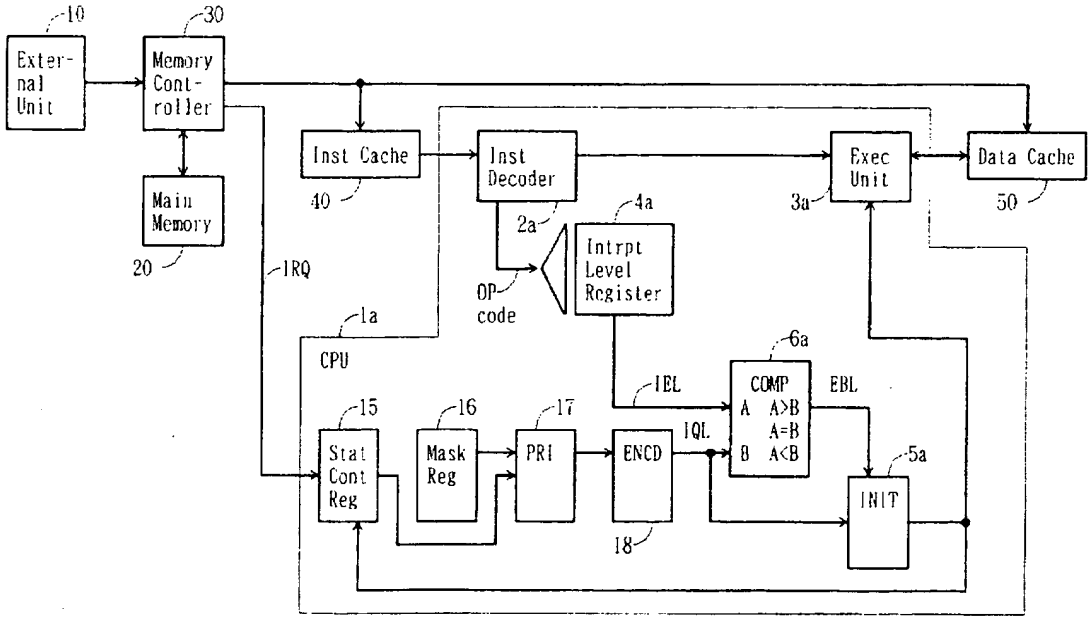
4,349,873	9/1982	Fukasawa et al.	395/742
5,163,152	11/1992	Okamoto	395/733
5,168,566	12/1992	Kuki et al.	395/650
5,212,779	5/1993	Saito	395/375
5,212,796	5/1993	Allison	395/737
5,218,703	6/1993	Fleck et al.	395/737
5,222,215	6/1993	Chou et al.	395/735
5,222,239	6/1993	Rosch	395/750
5,257,357	10/1993	Yishay et al.	395/725
5,261,107	11/1993	Klim et al.	395/725
5,280,618	1/1994	Takagi	395/725
5,301,283	4/1994	Thacker et al.	395/325
5,301,331	4/1994	Ueno et al.	395/725
5,313,640	5/1994	Beardsley et al.	395/725
5,410,715	4/1995	Ishimoto et al.	395/775
5,495,615	2/1996	Nazar et al.	395/733

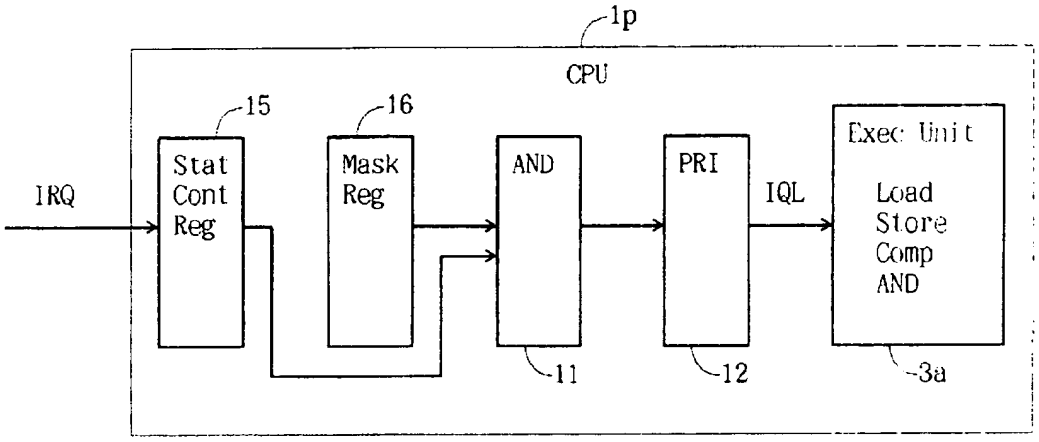
Primary Examiner—Jack B. Harvey
Assistant Examiner—Paul R. Myers
Attorney, Agent, or Firm—Staas & Halsey

[57] **ABSTRACT**

An interrupt control circuit, used for a data processing unit which fetches and executes an instruction and which has an interrupt function, for controlling an interrupt request signal which is input from a device external to the data processing unit and which has a respective interrupt level assigned. The interrupt control circuit stores in an interrupt level register, interrupt-accept level data which corresponds to the instruction and specifies the interrupt level of the interrupt request signal to be accepted for interrupt; reads the interrupt-accept level data corresponding to the instruction from the interrupt level register when an instruction is fetched for execution and compares the interrupt-accept level read with the interrupt level of the interrupt request signal; and accepts the interrupt request signal to initiate the interrupt function of the data processing unit in dependence upon a result of the comparison.

14 Claims, 9 Drawing Sheets





PRIOR ART

FIG. 1

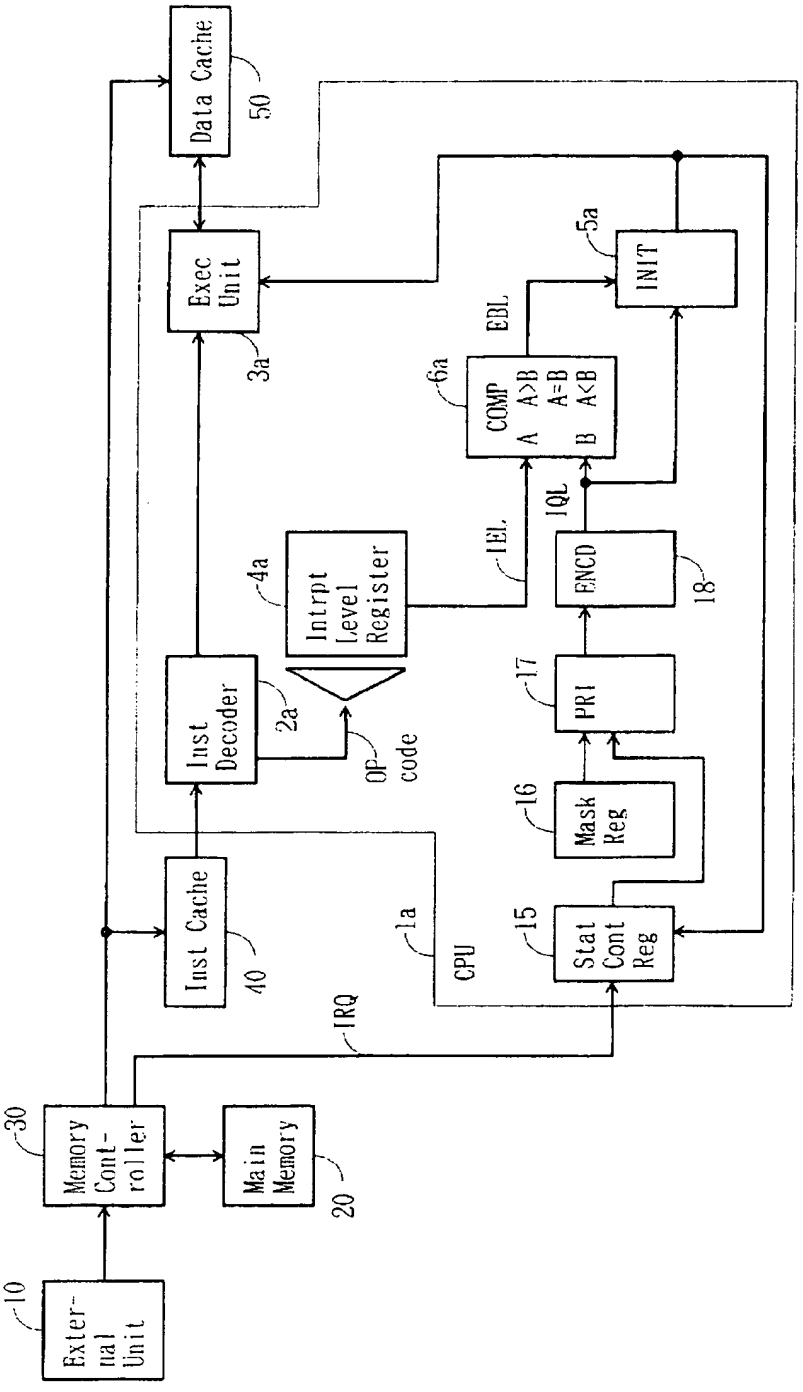

Data Cache
50
Exec Unit
3a
5a
INIT
EBL
6a
COMP
A A>B
A=B
B A<B
IQL
IEL
4a
Intrpt Level Register
ENCD
18
Inst Decoder
2a
OP code
17
PRI
16
Mask Reg
15
Stat Cont Reg
Inst Cache
40
1a
CPU
IRQ
30
Memory Cont-roller
Main Memory
20
10
Exter-nal Unit

FIG. 2A

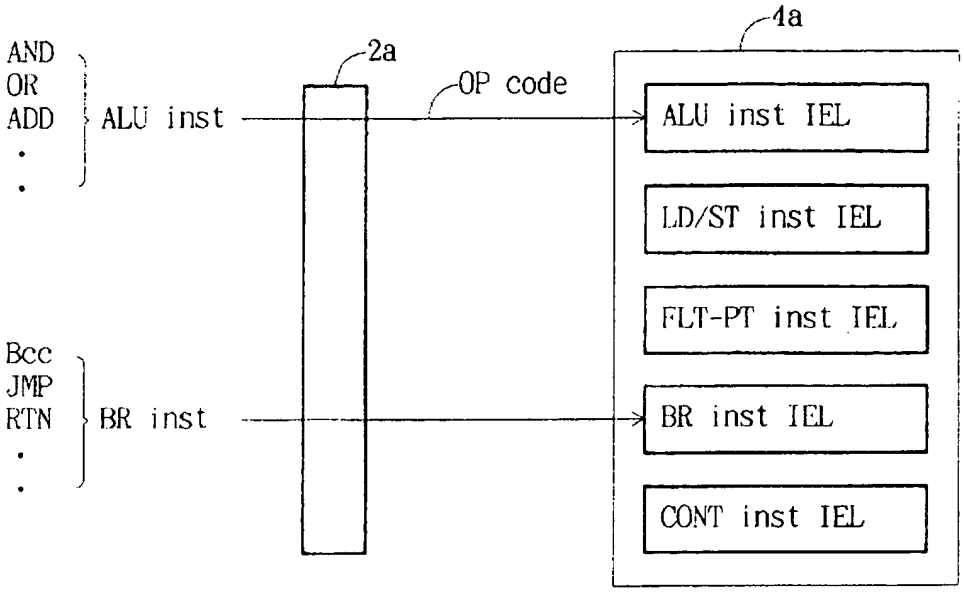

AND
OR
ADD
ALU inst
2a
OP code
4a
ALU inst IEL
LD/ST inst IEL
FLT-PT inst IEL
Bcc
JMP
RTN
BR inst
BR inst IEL
CONT inst IEL

FIG. 2B

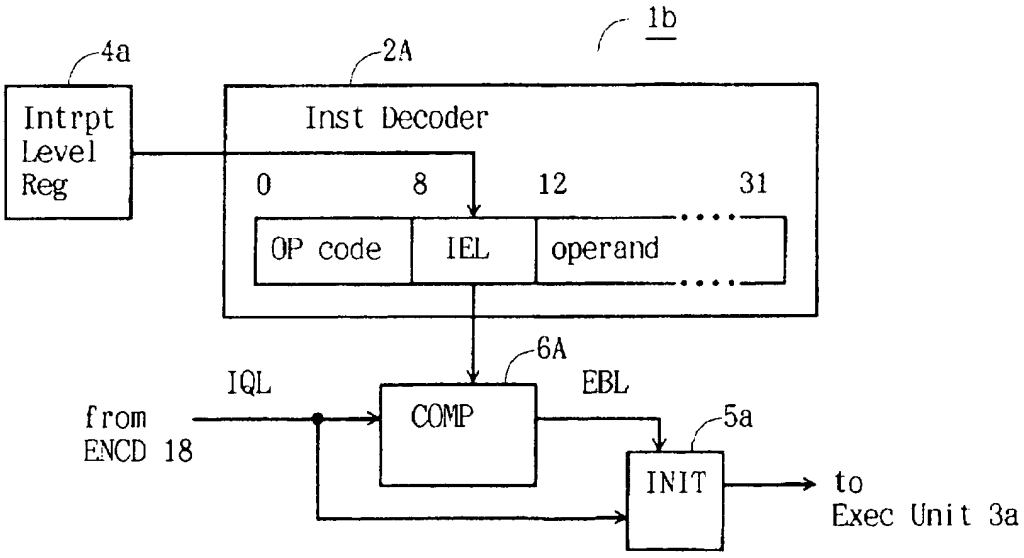

1b
4a
2A
Intrpt
Level
Reg
Inst Decoder
0
8
12
31
OP code
IEL
operand
6A
IQL
EBL
from
ENCD 18
COMP
5a
INIT
to
Exec Unit 3a

FIG. 2C

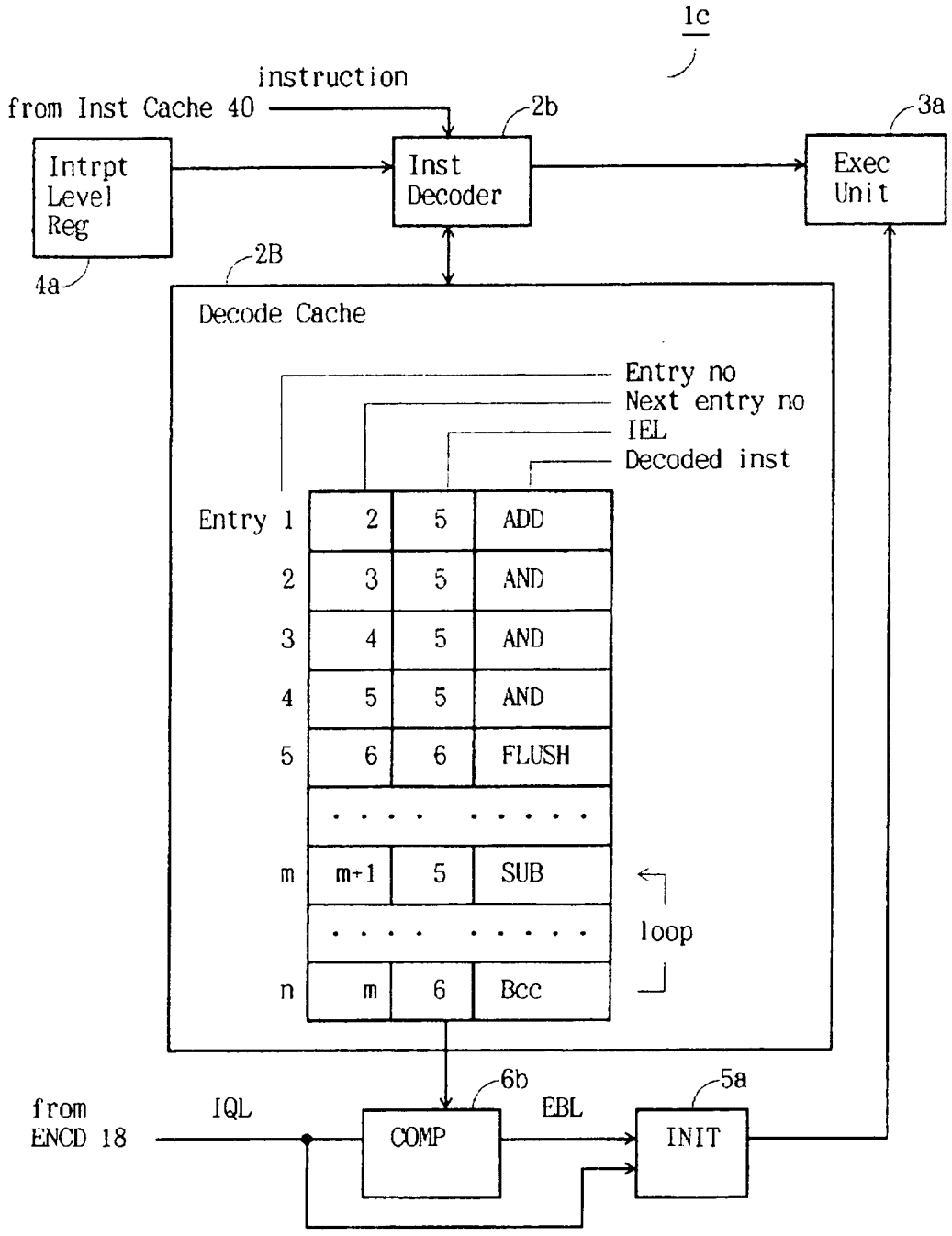

1c
instruction
from Inst Cache 40
2b
3a
Intrpt Level Reg
Inst Decoder
Exec Unit
4a
2B
Decode Cache
Entry no
Next entry no
IEL
Decoded inst
Entry 1
2 5 ADD
2 3 5 AND
3 4 5 AND
4 5 5 AND
5 6 6 FLUSH
m m+1 5 SUB
loop
n m 6 Bcc
6b
5a
from ENCD 18
IQL
COMP
EBL
INIT

FIG. 3

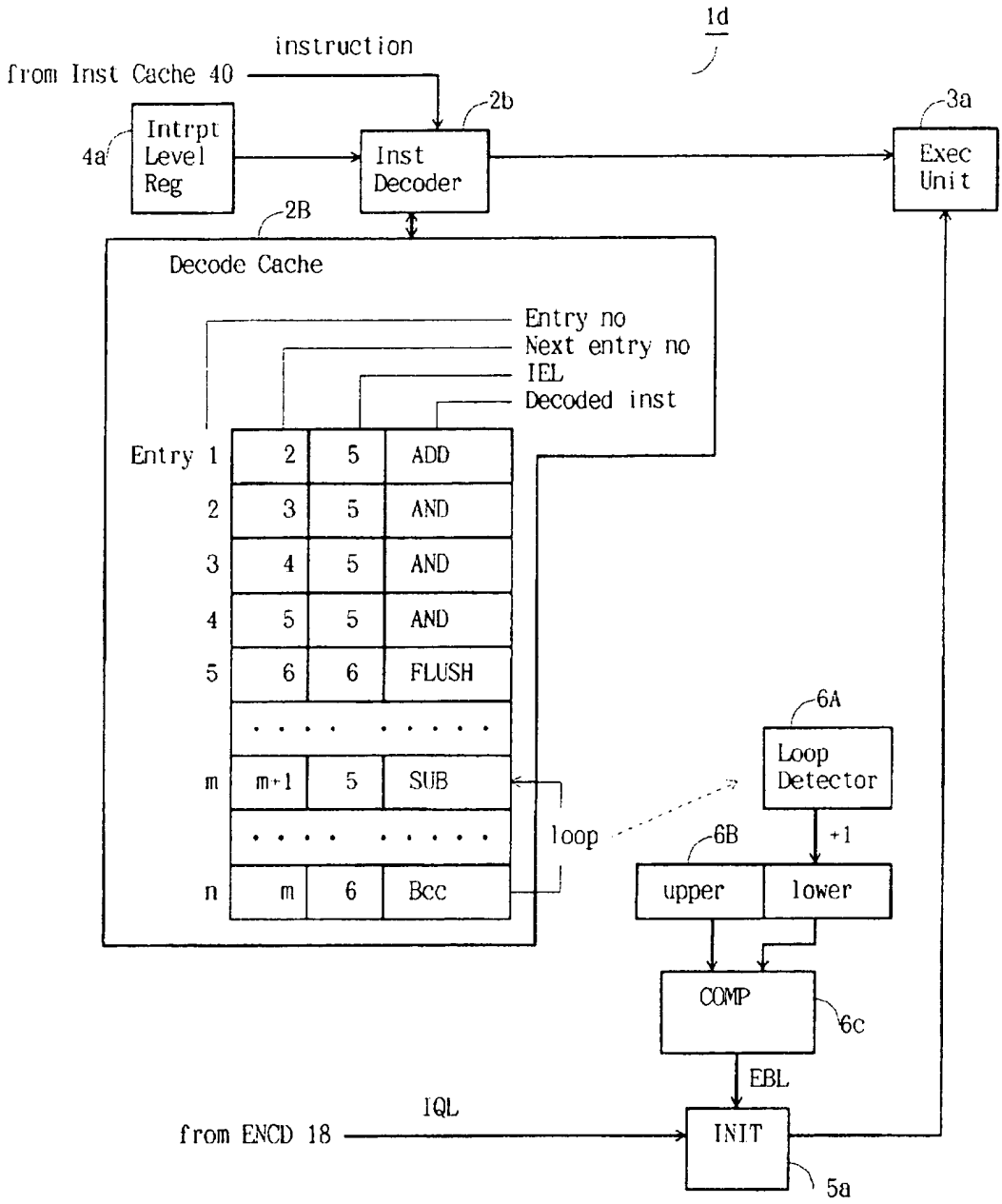

1d
instruction
from Inst Cache 40
2b
3a
4a
Intrpt Level Reg
Inst Decoder
Exec Unit
2B
Decode Cache
Entry no
Next entry no
IEL
Decoded inst
Entry 1
2
5
ADD
2
3
5
AND
3
4
5
AND
4
5
5
AND
5
6
6
FLUSH
m
m+1
5
SUB
loop
n
m
6
Bcc
6A
Loop Detector
+1
6B
upper
lower
COMP
6c
EBL
IQL
from ENCD 18
INIT
5a

FIG. 4

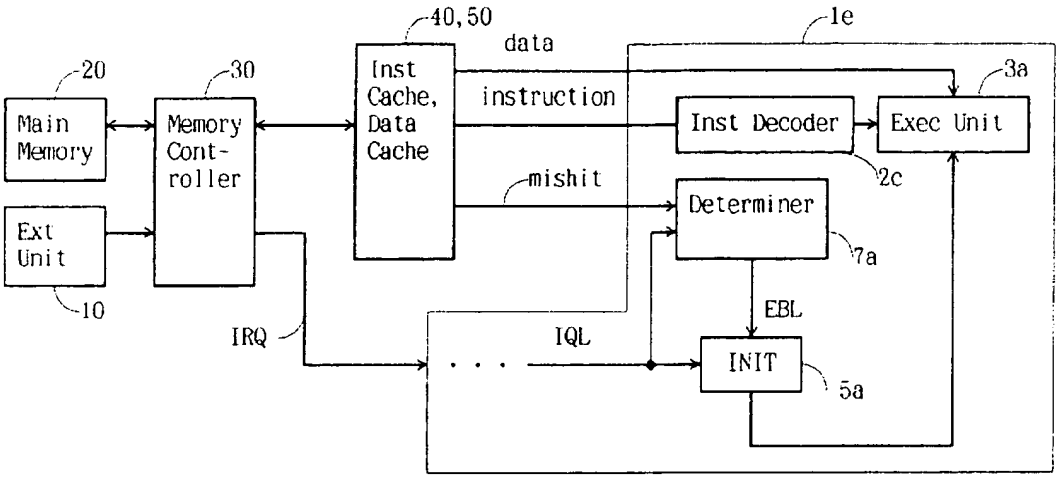

40,50
data
1e
20
30
Inst
Cache,
Data
Cache
instruction
3a
Main
Memory
Memory
Cont-
roller
Inst Decoder
Exec Unit
2c
mishit
Determiner
Ext
Unit
7a
10
EBL
IRQ
IQL
INIT
5a

FIG. 5A

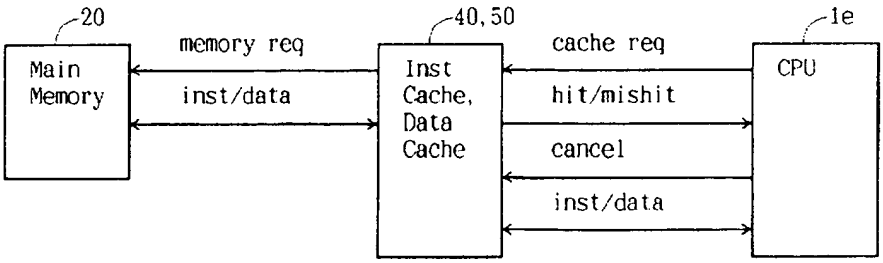

20
memory req
40,50
cache req
1e
Main
Memory
inst/data
Inst
Cache,
Data
Cache
hit/mishit
CPU
cancel
inst/data

FIG. 5B

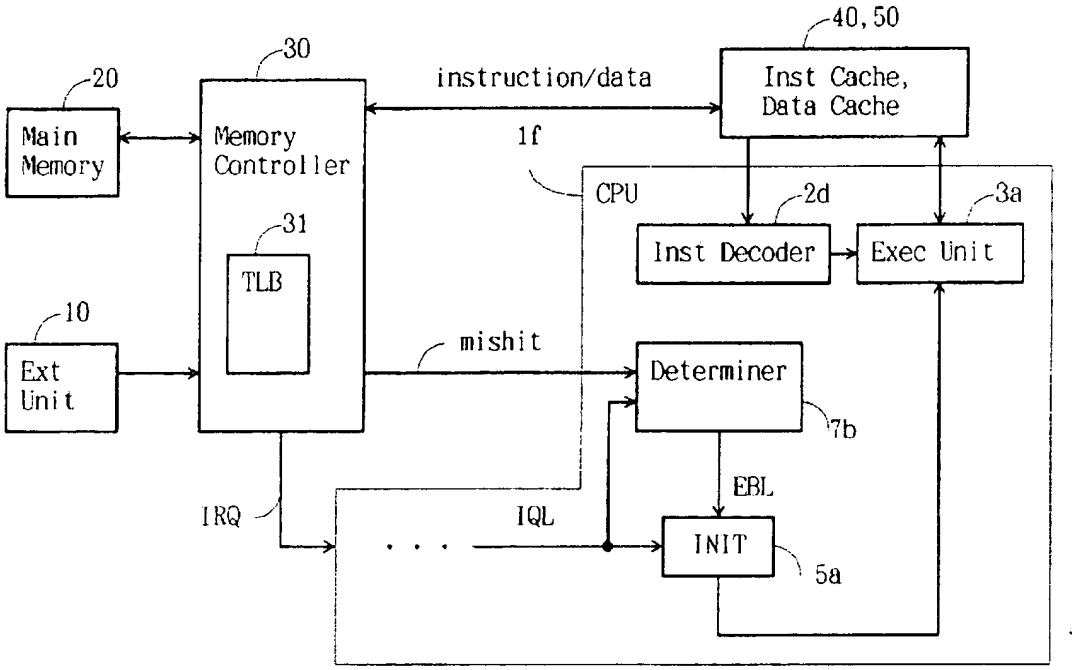

40, 50
Inst Cache,
Data Cache
30
20
instruction/data
Main
Memory
Memory
Controller
1f
CPU
2d
3a
Inst Decoder
Exec Unit
31
TLB
10
Ext
Unit
mishit
Determiner
7b
EBL
IRQ
IQL
INIT
5a

FIG. 6A

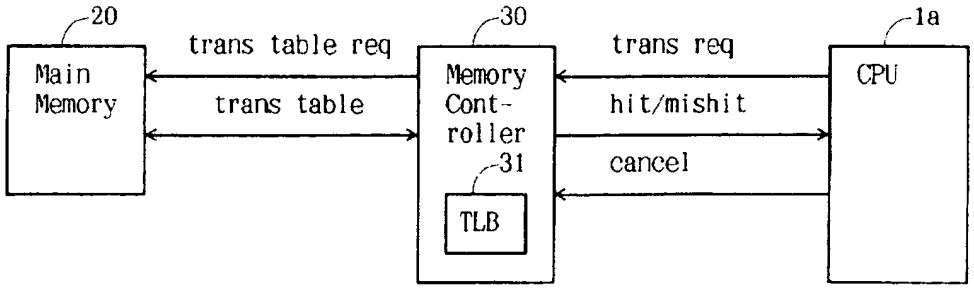

20
30
1a
trans table req
trans req
Main
Memory
Memory
Cont-
roller
CPU
trans table
hit/mishit
cancel
31
TLB

FIG. 6B

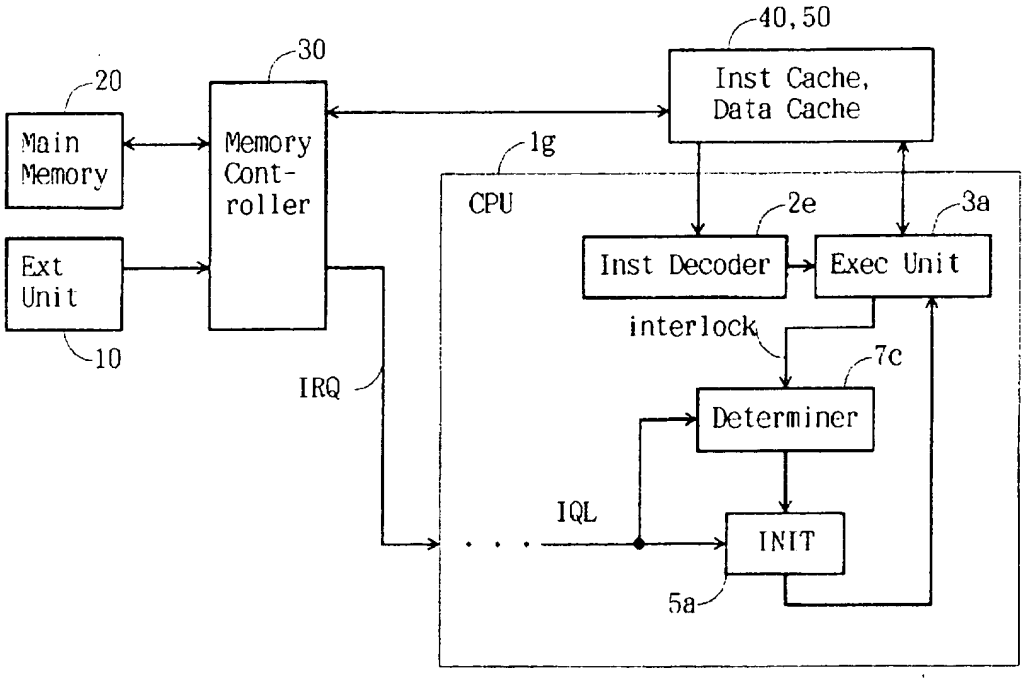

40,50
Inst Cache,
Data Cache
30
20
Main
Memory
Memory
Cont-
roller
1g
CPU
2e
3a
Inst Decoder
Exec Unit
Ext
Unit
10
interlock
7c
IRQ
Determiner
IQL
INIT
5a

FIG. 7A

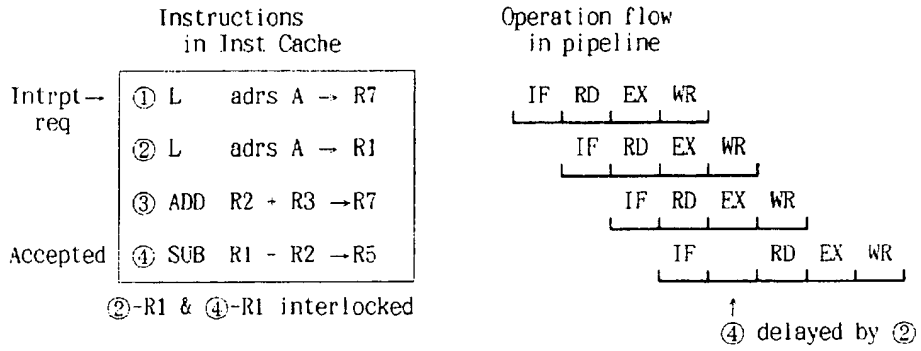

Instructions
in Inst Cache
Intrpt→
req
① L adrs A → R7
② L adrs A → R1
③ ADD R2 + R3 →R7
Accepted
④ SUB R1 - R2 →R5
②-R1 & ④-R1 interlocked
Operation flow
in pipeline
IF RD EX WR
IF RD EX WR
IF RD EX WR
IF RD EX WR
④ delayed by ②

FIG. 7B

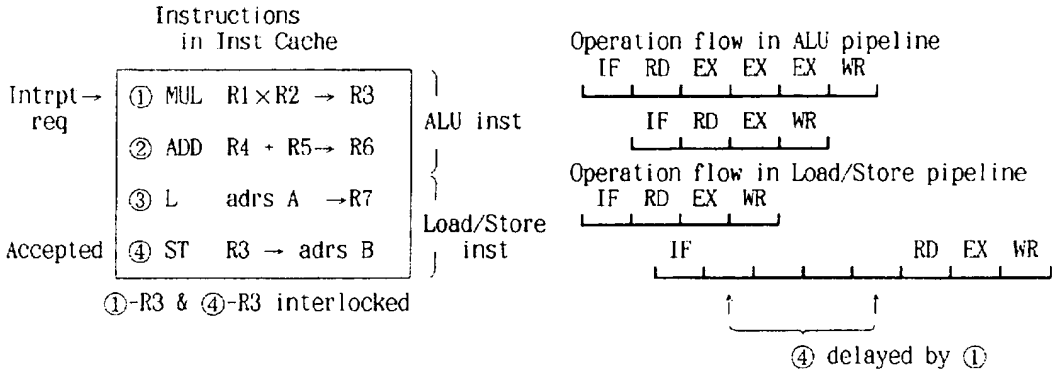

Instructions
in Inst Cache
Intrpt→
req
① MUL R1×R2 → R3
② ADD R4 + R5→ R6
③ L adrs A →R7
Accepted
④ ST R3 → adrs B
ALU inst
Load/Store
inst
①-R3 & ④-R3 interlocked
Operation flow in ALU pipeline
IF RD EX EX EX WR
IF RD EX WR
Operation flow in Load/Store pipeline
IF RD EX WR
IF RD EX WR
④ delayed by ①

FIG. 7C

INTERRUPT CONTROL CIRCUIT

BACKGROUND OF THE INVENTION

1. Field of the Invention

The present invention is related to an interrupt control circuit of a data processing unit, especially, to an interrupt control circuit which accepts or holds an external interrupt request signal (or an interrupt request signal from an unit external to the data processing unit).

In recent years, many data processing units usually improve their processing performance by utilizing special facilities such as a pipeline and various types of buffers including cache memories, for instruction and data. Data processing units having such facilities can demonstrate increased processing performance when an external interrupt, e.g., an input/output interrupt and timer interrupt, is not triggered. When an external interrupt is triggered, performance decreases because it may invalidate instructions and data being processed in those facilities.

Accordingly, an interrupt control circuit which allows a data processing unit to demonstrate an increased processing performance, even when an external interrupt is triggered, is in great demand.

2. Description of the Related Art

FIG. **1** is a block diagram illustrating an interrupt control circuit as is known in the prior art.

External interrupt (e.g., input/output interrupt and timer interrupt) request signals IRQ, with respective priority levels (e.g., 1–15) assigned, are first set in a state control register **15**. The outputs from the state control register **15** are ANDed with interrupt-mask information (mask bits), which are set by a program in a mask register **16**, by an AND circuit **11**. The mask bits, corresponding to the external interrupt requests, specify whether to enable the corresponding interrupt requests. The interrupt request signal whose corresponding mask bit is set to 1 is selected by the AND circuit **11** to output logical **1** to a priority circuit **12**. When a plurality of interrupt request signals are selected by the AND circuit **11**, the priority circuit **12** selects only one of the highest priority level to enable for interrupt.

The interrupt request signal IQL is input to an execution unit **3***a*. When fetching, or after having fetched, an instruction from a programmed instruction stream, the execution unit **3***a* checks to see if there is an interrupt request signal IQL enabled. If there is, the execution unit **3***a* accepts the interrupt request and performs an interrupt function; otherwise, it fetches an instruction and executes the instruction fetched.

According to the related art as described above, when an external interrupt request IRQ is input while a data processing unit (hereinafter called a CPU) **1***p* is executing an instruction stream, the external interrupt request IRQ is accepted or held, simply depending on the interrupt-mask information stored in the mask register **15**. That is, when the above condition of the interrupt-mask information is satisfied for an interrupt request, the CPU **1***p* accepts the interrupt request and performs interrupt operation, even when the interrupt request is of lower priority, i.e., not so urgent to interrupt the program immediately at the instruction being executed when the interrupt request is input.

Eventually, instructions and data in a cache memory and also instructions in a pipeline, which are all provided to achieve a high-speed processing performance of CPU **1***p*, become invalid and have to be discarded.

SUMMARY OF THE INVENTION

It is an object of the present invention to provide an interrupt control circuit which allows a data processing unit to demonstrate an increased processing performance even when an external interrupt request are generated.

To achieve the above and other objects, the present invention provides a level storage device, a comparison device and an acceptance device.

In an interrupt control circuit, used for a data processing unit which fetches and executes an instruction and which has an interrupt function for controlling an interrupt request signal which is input from a device external to the data processing unit and which has a respective interrupt level assigned, the level storage device stores interrupt accept level data which corresponds to the instruction and specifies the interrupt level of the interrupt request signal to be accepted for interrupt The comparison device reads the interrupt-accept level data corresponding to the instruction from the level storage means and compares the interrupt-accept level read with the interrupt level of the interrupt request signal when an instruction is fetched for execution. The acceptance device accepts the interrupt request signal to initiate the interrupt function of the data processing unit in dependence upon the comparing by the first comparison device.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1** is a block diagram illustrating an interrupt control circuit of the related art;

FIG. **2**A is a block diagram illustrating an interrupt control circuit according to a first embodiment of the present invention;

FIG. **2**B is a schematic diagram illustrating an interrupt level register of the present invention;

FIG. **2**C is a schematic diagram illustrating a modification of the first embodiment of the present invention;

FIG. **3** is a schematic diagram illustrating an interrupt control circuit according to a second embodiment of the present invention;

FIG. **4** is a schematic diagram illustrating an interrupt control circuit according to a third embodiment of the present invention;

FIG. **5**A is a block diagram illustrating an interrupt control circuit according to a fourth embodiment of the present invention;

FIG. **5**B is a schematic diagram illustrating a cache mishit;

FIG. **6**A is a block diagram illustrating an interrupt control circuit according to a fifth embodiment of the present invention;

FIG. **6**B is a schematic diagram illustrating a TLB mishit;

FIG. **7**A is a block diagram illustrating an interrupt control circuit according to a sixth embodiment of the present invention; and

FIGS. **7**B and **7**C are schematic diagrams illustrating a pipeline interlock.

Throughout the above-mentioned drawings, identical reference numerals are used to designate the same or similar component parts.

DESCRIPTION OF THE PREFERRED EMBODIMENTS

FIG. **2**A is a block diagram illustrating an interrupt control circuit according to the first embodiment of the present invention.

The instruction set of a data processing unit (CPU) is divided broadly into general instructions and control instructions. The control instructions, which manipulate the CPU hardware itself and its facilities, stop and invalidate pipeline operations temporarily and invalidate the data in a cache memory (an instruction cache **40** and an data-cache **50** in the Figure).

The interrupt control circuit of the first embodiment of the present invention holds an external interrupt request while the general instructions are being executed until a control instruction appears and accepts the interrupt request on detecting, in the process of executing an instruction stream, a control instruction which prevents the data processing unit from making the best use of it's high-speed processing features and facilities. Hereinafter, a control instruction is called an interruptible instruction in this sense.

Looking at FIG. 2A, an external unit **10** is, for example, an input/output unit or a timer. The input/output unit transfers (writes/reads) data to/from a main memory **20** through a memory controller **30** and generates an external interrupt request signal when completing a transfer of a predesignated amount of data, for example. The timer generates an interrupt request when, for example, a predesignated length of time has elapsed. The memory controller **30** arbitrates a conflict between two or more input/output devices requesting data transfer to/from the main memory **20**. It also outputs an interrupt request (IRQ) signal to the CPU **1***a* in accordance with an interrupt request signal from the external unit **10**.

External interrupt (e.g., input/output interrupt and timer interrupts) request signals, with respective priority levels (e.g., 1–15) assigned, are first set in a state control register **15**. A mask register **16** stores interrupt mask bits, which are set by a program, to specify whether to enable the respective levels of interrupt requests. The interrupt request signals from the state control register **15** are each ANDed by a priority circuit (PRI) **17**. Corresponding interrupt mask bits and those having corresponding mask bits set to 1 are selected. Further, of the interrupt request signals selected, only the one of highest priority is selected by the priority circuit **17** and outputted to an encoder circuit (ENCD) **18**. There, the interrupt level of the selected interrupt request signal, for example, is encoded into a 4-bit IQL signal to represent 15 interrupt levels.

An instruction cache (Inst Cache) **40**, is provided to reduce time required for the CPU **1***a* to fetch instructions from the main memory **20**. Preferably the instruction cache **40** is a high-speed buffer storage for containing a copy of such instructions from the main memory **20** as are most probably to be used by CPU **1***a*. A data cache **50**, provided for the same purpose as the instruction cache **40**, is for containing a copy of data from the main memory **20**.

An interrupt level register (Intrpt Level) **4***a* contains interrupt-enable level information that is set by a program in order to specify interrupt levels corresponding to the kind of instructions for enabling an external interrupt request. The interrupt-enable level information each consists of, for example, 4 bits corresponding to the 15 interrupt levels of the external interrupt request.

An instruction decoder **2***a* fetches an instruction from the instruction cache **40** and decodes it in order to facilitate execution of the instruction by an execution unit (Exec Unit) **3***a*. It also refers to the interrupt level register **4***a* for the interrupt-enable level information based on an operation (OP) code of the instruction decoded, as shown in FIG. **2B**.

The execution unit **3***a* interprets the OP code of the instruction decoded and executes it according to the OP code. It also performs an interrupt function which stops a running program in such a way that it can be resumed at a later time, and in the meanwhile permits other program to be executed.

FIG. **2B** is a schematic diagram illustrating an interrupt level register of the present invention.

For ALU instructions, such as AND, OR and ADD, which use a hardware arithmetic and logical unit (ALU), an ALU instruction part of the interrupt level register **4***a* is referred to for the interrupt-enable level (IEL) data. Similarly, for branch instructions, such as conditional branch (Bcc), jump (JMP) and return (RTN) instructions, a branch instruction part of interrupt level register **4***a* is referred to.

A comparator (COMP) **6***a* compares the selected interrupt-enable level (IEL) with the interrupt level of the interrupt request IQL from the encoder **18** (not shown, see FIG. 2A). Depending upon the comparison result, the comparator **6***a* outputs an EBL signal indicating that the interrupt request IQL is to be enabled for interrupt. On receipt of the EBL signal, an initiator (INIT) **5***a* gates out 4 bits, for example, of the interrupt level of the external interrupt request IQL to the execution unit **3***a* to initiate the interrupt function of the execution unit **3***a*.

For example, it is assumed that there are a total of six interrupt levels 0–5 of the external interrupt requests, with level **0** having the highest priority and level **5** having the lowest priority. It is further assumed that the interrupt request is to be accepted when the interrupt level of the external interrupt request signal is smaller than the interrupt-enable level in the interrupt level register **4***a*, wherein the interrupt-enable level corresponds to the OP code of the fetched instruction.

In this example, if an interrupt request signal having interrupt level **5** is input as the IQL signal via the memory controller **30** and the circuits **15–18** when the instruction decoder **2***a* has fetched an instruction having interrupt-enable level **5** according to the interrupt level register **4***a*, that external interrupt request is not accepted, but held in accordance with the above condition. When an instruction having interrupt-enable level **6** or more appears, the interrupt request being held is enabled for interrupt and accepted.

FIG. **2C** is a schematic diagram illustrating a modification of the first embodiment of the present invention.

Using this modification, when decoding an instruction, an instruction decoder **2A** reads from the interrupt level register **4***a*, the interrupt-enable level (IEL) data corresponding to the OP code and adds the level data to the OP code. A comparator **6A** then compares the interrupt-enable level (IEL) in the instruction decoder **2A** with that of the interrupt request IQL from the encoder circuit **18** (see FIG. 2C) to determine whether to enable or disable the interrupt request IQL in dependence upon the comparison result. This produces the same effect as in the above example.

FIG. **3** is a schematic diagram illustrating an interrupt control circuit according to the second embodiment of the present invention, in which the CPU **1***c* is provided with a decode cache **2B**.

The instruction decoder **2***b* fetches a plurality of instructions from the instruction cache **40**, decodes them to facilitate execution of the instructions by the execution unit **3***a*, and stores the decoded instruction in the decode cache **2B**. When the instructions form a loop in the decode cache **2B**, as shown in FIG. **3**, the looped instructions need not be fetched from the instruction cache **40**, repeatedly every time a branch instruction is encountered and therefore, the looped instructions can be executed at a high speed. Thus, a decode cache is effective and is often used in a data processing unit.

The decode cache **2B** comprises a plurality of (n) entries, each entry including at least a decoded instruction (for easy understanding, only the OP code is shown) and a next entry number specifying the entry whose instruction is to be executed next. In the example shown in FIG. **2C**, when decoding the instructions fetched from the instruction cache **40**, the instruction decoder **2***b* reads the interrupt-enable level data (IEL), corresponding to the OP code of the instructions, from the interrupt level register **4***a*, and stores the level data in each entry.

If there is an interrupt request input or held when the instruction decoder **2***b* fetches a decoded instruction from the decode cache **2B** to have the execution unit **3***a* execute it, a comparator (COMP) **6***b* compares the interrupt level of the external interrupt request IQL from the encoder **18** (not shown, see FIG. **2A**) with the interrupt-enable level (IEL) of the instruction fetched.

For example, when the former is smaller than the latter, the comparator **6***b* outputs an EBL signal indicating that the interrupt request IQL is to be enabled for interrupt. On receipt of the EBL signal, an initiator (INIT) **5***a* gates out, for example, 4 bits signifying the interrupt level of the external interrupt request IQL to the execution unit **3***a* to initiate the interrupt function of the execution unit **3***a*.

In the following example, too, it is assumed that there are a total of six interrupt levels **0–5** of external interrupt requests, with level **0** having the highest priority and level **5** having the lowest priority, and that the interrupt request IQL is enabled on condition that the interrupt level of the external interrupt request signal IQL is smaller than the interrupt-enable level stored in the decode cache **2B** along with the instruction.

It is also assumed that an interrupt request signal IRQ input to the CPU **1***c* has interrupt level **5** and that it is selected and encoded into the IQL signal via the circuits **15–18**, as shown in FIG. **2A**. At this point, CPU **1***c* does not immediately accept the interrupt request, but checks the decode cache **2B** for an interruptible instruction, i.e., control instruction and holds the interrupt request until the interruptible instruction is executed.

In this example, the external interrupt request is held because the interrupt level **5** of the interrupt request in not smaller than the interrupt-enable level **5** of the first four general instructions. Then, when the fifth instruction (FLUSH) is encountered, the interrupt request which has been held is accepted, because the interrupt level **5** of the interrupt request is smaller than the interrupt-enable level **6** of the FLUSH instruction. The FLUSH instruction, which is defined for hardware to manipulate the cache memory, writes the contents of the instruction cache **40** (or data cache **50**) into the main memory **20** so as to keep the contents of both memories matched.

FIG. **4** is a schematic diagram illustrating an interrupt control circuit according to the third embodiment of the present invention, in which the CPU **1***d* is provided with a loop counter **6B** in addition to the elements shown as part of CPU **1***c* in FIG. **3**. The upper part of the loop counter **6B** stores an upper bound, previously set by a program, loop repetitions and the lower part counts the loops executed.

When the CPU **1***d* has fetched an entry (e.g., entry n in the example) from the decode cache **2B** for execution, a loop detector **6A** checks to see if the next entry number (e.g., entry m in the example) thereof, which indicates the entry to be executed next, is smaller than the present entry number (n). If so, the loop detector **6A** determines that the program forms a loop and, when there is an interrupt request input or held, adds one to the lower part of the loop counter **6B**. A comparator (COMP) **6***c* compares the contents of upper and lower parts of the loop counter **6B** and, when both become equal, outputs an EBL signal indicating that the interrupt request IQL is to be enabled for interrupt. The comparator **6***c* then clears the lower part to zero.

On receipt of the EBL signal, an initiator (INIT) **5***a* gates out, for example, 4 bits of the interrupt level of the external interrupt request IQL to the execution unit **3***a* to initiate the interrupt function of the execution unit **3***a*. This can prevent such inconveniences as a time-out error from occurring when an interrupt request is not accepted but held for a long time because of a program loop including no interruptible instruction.

FIG. **5A** is a block diagram illustrating an interrupt control circuit according to the fourth embodiment of the present invention. FIG. **5B** is a schematic diagram illustrating a cache mishit.

A CPU **1***e* sends a "cache request" signal to the instruction cache **40** (or data cache **50**) to request an instruction (or data). If the instruction cache **40** does not include the instruction requested, it sends a "mishit" signal to the CPU **1***e* and reads the instruction (or data) from the main memory **20**, instead. However, the time required for reading the instruction (or data) from the main memory **20** is extremely long compared with a machine cycle of the CPU **1***e*. The present invention saves that time by performing an interrupt operation instead of fetching the instruction (or data) from the main memory **20**.

If there is an external interrupt request input and enabled as the IQL signal when the mishit signal is received, a determiner **7***a* causes the CPU **1***e* to send a "cancel" signal to the instruction cache **40** so as to cancel the cache request signal and outputs an EBL signal indicating that the interrupt request IQL is to be enabled for interrupt. On receipt of the EBL signal, an initiator (INIT) **5***a* gates out the 4-bit interrupt level signal of the external interrupt request IQL to the execution unit **3***a* to initiate the interrupt function of the execution unit **3***a*. Unless there is an external interrupt request input when the mishit signal is received, the CPU **1***e* does not send the cancel signal but keeps sending the cache request signal to read the instruction from the main memory **20**.

FIG. **6A** is a block diagram illustrating an interrupt control circuit according to the fifth embodiment of the present invention. FIG. **6B** is a schematic diagram illustrating a TLB mishit.

This embodiment is associated with an interrupt control circuit for an CPU If with a dynamic address translation feature which, in a virtual memory system, changes a virtual memory address to a real memory address during execution of an instruction.

To address-translates an instruction or data, CPU **1***f* sends a "translation request" signal (abbreviated to "trans request" in the Figure) to the memory controller **30**. If a translation table for translating the instruction or data is not in a special buffer called Translation Lookaside Buffer (abbreviated to TLB) **31**, the memory controller **30** sends a "mishit" signal to CPU If and reads the translation table from the main memory **20**. It then performs an address translation using the translation table read from the main memory **20** and stores the translation table in the TLB **31**. The time required for reading the translation table from the main memory **20** is extremely long compared with a machine cycle of the CPU **1***f*. The present invention saves that time by performing an interrupt operation instead of reading the translation table from the main memory **20** as follows:

In the same way as the above example, if there is an external interrupt request input and enabled as the IQL signal when the mishit signal is received, a determiner **7***b* causes the CPU If to send a "cancel" signal to the the memory controller **30** so as to cancel the translation request signal and outputs an EBL signal indicating that the interrupt request IQL is to be enabled for interrupt. On receipt of the EBL signal, an initiator (INIT) **5***a* gates out the 4-bit interrupt level of the external interrupt request IQL to the execution unit **3***a* to initiate the interrupt function of the execution unit **3***a*. Unless there is an external interrupt request input when the mishit signal is received, the CPU **1***f* is not caused to send the cancel signal but keeps sending the translation request signal to read the translation table from the main memory **20**.

FIG. **7**A is a block diagram illustrating an interrupt control circuit of the sixth embodiment of the present invention. FIGS. **7**B and **7**C are schematic diagrams illustrating a pipeline interlock.

In recent years, data processing units utilizing a pipeline or even a plurality of pipelines for execution of instructions in order to reinforce processing performance is wide-spread.

When different instructions within a pipeline or between a plurality of pipelines share the same single hardware resource, they may interfere with each other, causing an interlock. The interlock prevents an instruction from being executed in step with a system clock, causing a delay in execution of the instruction. The present invention accepts an external interrupt request, if any, when an interlock is developed and performs an interrupt operation, thus making use of the time in which the instruction, i.e., program execution is delayed.

In FIG. **7**A, when detecting an interlock developed in a pipeline or between pipelines, an execution unit **3***a* outputs an interlock signal to a determiner **7***c*. The determiner **7***c* checks to see if there is an external interrupt request is input and enabled as the IQL signal and, if there is, sends an EBL signal indicating that the interrupt request IQL is to be enabled for interrupt. Upon receipt of the EBL signal, an initiator (INIT) **5***a* gates out the 4-bit interrupt level signal of the external interrupt request IQL to the execution unit **3***a* to initiate the interrupt function of the execution unit **3***a*.

FIG. **7**B shows an example of an interlock developed in a single pipeline. The righthand figure shows a fashion in which instructions **1**–**4** listed left flow in a pipeline and an interlock develops therein.

Instruction **1** loads contents of address A into register R**7**. Instruction **2** loads contents of address A into register R**1**. Instruction **3** adds contents of register R**2** to those of R**3** and stores the addition result in register R**7**. Instruction **4** subtracts contents of register R**3** from those of register R**1** and stores the subtraction result in register R**7**.

IF, RD, EX, and WR at pipeline stages represent instruction fetching, operand reading, instruction execution (e.g., addition, subtraction, and multiplication), and operand storing, respectively. In the Figure, instruction **2** and instruction **4** fall in an interlock in an attempt to access register R**1**, which causes instruction **4** to wait at the pipeline stage RD until instruction **2** is completed. Accordingly, an external interrupt request, which is generated at the timing the instruction **1** is fetched, is held until the interlock is detected during execution of instruction **4** and is accepted immediately thereafter.

FIG. **7**C shows an example of an interlock developed between a plurality of pipelines.

The righthand figure shows a fashion in which instructions **1**–**4** listed left flow in two pipelines (ALU pipeline for instructions which use the ALU; Load/Store pipeline for load/store instructions) and an interlock develops therebetween.

In the Figure, instruction **1** and instruction **4** fall in an interlock in an attempt to access register RS, which causes instruction **4** to wait at the pipeline stage RD until instruction **1** is completed. Accordingly, an external interrupt request, which is generated at the timing the instruction **1** is fetched, is held until the interlock is detected during execution of instruction **4**, and accepted immediately thereafter.

As described above, the present invention specifies interrupt-enable levels specifying, corresponding to the kind of instructions, what level of an external interrupt request to accept for interrupt. If there is an interrupt request input when an instruction is fetched, the interrupt-enable level corresponding to the instruction is compared with the level of the interrupt request and the interrupt request is accepted or held in dependence upon the comparison result. Also, the present invention accepts an interrupt request when desired item of data is not found in a cache memory or translation lookaside buffer or when an interlock develops in s pipeline.

Accordingly, the present invention can solve a problem of the conventional interrupt control circuit which unconditionally accepts an external interrupt request enabled by a mask bit, even if it may not be so urgent, eventually invalidating the instructions and data in a cache memory or a pipeline. Therefore, the present invention can make the best use of the high-speed processing features and facilities and therefore, can have the data processing unit demonstrate an expected processing performance.

What is claimed is:

1. An interrupt control circuit for a data processing unit which fetches and executes instructions for controlling an interrupt request signal from an external device which has an assigned interrupt level, said interrupt control circuit comprising:

level storage means for storing interrupt-accept level data corresponding to each instruction and specifying the interrupt level of the interrupt request signal to be accepted for interrupt;

first comparison means for reading, when an instruction is fetched for execution, the interrupt-accept level data corresponding to the instruction from said level storage means and for comparing the interrupt-accept level with the interrupt level of the interrupt request signal; and

first acceptance means for accepting the interrupt request signal to initiate the interrupt function upon the comparing by said first comparison means.

2. An interrupt control circuit according to claim **1**, wherein said level storage means stores the interrupt-accept level data which corresponds to a classification of the instruction.

3. An interrupt control circuit according to claim **1**, wherein the interrupt-accept level data is set in said level storage means by a program.

4. An interrupt control circuit according to claim **1**, further comprising:

setting means for setting a number;

loop detection means for detecting a loop of the instructions during execution of the instructions;

count means for counting the loops detected by said loop detection means;

second comparison means for comparing the number set by said setting means with the number of loops counted by said count means; and

second acceptance means for accepting the interrupt request signal to initiate the interrupt function of the data processing unit upon the comparing by said second comparison means.

5. An interrupt control circuit for a data processing unit which has first buffer storage means for storing a plurality of instructions, the interrupt control circuit for controlling an interrupt request signal which is input from an external device which has an assigned interrupt level, said interrupt control circuit comprising:

level storage means for storing interrupt-accept level data corresponding to an instruction and specifying the interrupt level of the interrupt request signal to be accepted for interrupt;

transfer means for reading the interrupt-accept level data, from said level storage means, corresponding to the instruction stored in the first buffer storage means and for storing the interrupt-accept level data into an area of the first buffer storage means corresponding to the instruction;

first comparison means for reading the interrupt-enable level data, from the first buffer storage means, corresponding to an instruction fetched from the first buffer storage means for execution and for comparing the interrupt-enable level with the interrupt level of the interrupt request signal; and

first acceptance means for accepting the interrupt request signal to initiate the interrupt function of the data processing unit upon the comparing by said first comparison means.

6. An interrupt control circuit according to claim **5**, wherein said first buffer storage means stores a single instruction.

7. An interrupt control circuit according to claim **5**, wherein said level storage means stores the interrupt-accept level data which corresponds to a classification of the instruction.

8. An interrupt control circuit according to claim **5**, wherein the interrupt-accept level data is set in said level storage means by a program.

9. An interrupt control circuit according to claim **5**, further comprising:

setting means for setting a number;

loop detection means for detecting a loop of the instructions during execution of the instructions;

count means for counting the loops detected by said loop detection means;

second comparison means for comparing the number set by said setting means with the number of loops counted by said count means; and

second acceptance means for accepting the interrupt request signal to initiate the interrupt function of the data processing unit upon the comparing by said second comparison means.

10. An interrupt control circuit according to claim **9**, wherein

said first buffer storage means comprises a storage device which includes a plurality of entries, each entry having an entry number and storing an instruction and a next entry number which specifies the number of an entry whose instruction is to be executed next, and

said loop detection means detects a loop when an entry includes the next entry number which is smaller than the entry number thereof.

11. A method of controlling an interrupt request signal which is input from an external device having an assigned interrupt level during execution of instructions, said method comprising the steps of:

(a) storing interrupt-accept level data corresponding to each instruction which specifies the interrupt level of the interrupt request signal to be accepted for interrupt;

(b) reading the interrupt-accept level data corresponding to each instruction when an instruction is fetched for execution;

(c) comparing the interrupt-accept level read in step (b) with the interrupt level of the interrupt request signal; and

(d) accepting the interrupt request signal to initiate the interrupt function of the data processing unit in dependence upon the comparison in step (c).

12. A method according to claim **11**, further comprising:

(g) setting a number;

(h) detecting a loop of the instructions during execution of the instructions;

(i) counting the loops detected in step (h);

(j) comparing the number set in step (g) with the number of loops counted in step (i); and

(k) accepting the interrupt request signal to initiate the interrupt function of the data processing unit in dependence upon the comparison in step (j).

13. A method for using a data processing unit which has first buffer storage means for storing a plurality of instructions, to control an interrupt request signal input from an external device having an assigned interrupt level, said method comprising the steps of:

(a) storing interrupt-accept level data corresponding to the instruction specifying an interrupt level of the interrupt request signal to be accepted for interrupt;

(b) reading the interrupt-accept level data corresponding to the instruction;

(c) storing the interrupt-accept level data read in step (b) into an area which corresponds to the instruction;

(d) reading the interrupt-enable level data corresponding to a fetched instruction;

(e) comparing the interrupt-enable level read with the interrupt level of the interrupt request signal; and

(f) accepting the interrupt request signal to initiate the interrupt function of the data processing unit in dependence upon the comparing in step (e).

14. A method according to claim **13**, further comprising:

(g) setting a number;

(h) detecting a loop of the instructions during execution of the instructions;

(i) counting the loops detected in step (h);

(j) comparing the number set in step (g) with the number of loops counted in step (i); and

(k) accepting the interrupt request signal to initiate the interrupt function of the data processing unit in dependence upon the comparison in step (j).

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE

CERTIFICATE OF CORRECTION

PATENT NO. : 5,615,375
DATED : Mar. 25, 1997
INVENTOR(S) : IBUSUKI et al.

It is certified that error appears in the above-indentified patent and that said Letters Patent is hereby corrected as shown below:

Col. 1, line 14, after "buffers" insert --,--.

Col. 2, line 15, change "interrupt accept" to --interrupt-accept--.

Col. 5, line 59, after "program," insert --for--.

Col. 6, line 49, change "If" to --1f--;
line 59, change "If" to --1f--.

Col. 7, line 4, change "If" to --1f--.

Col. 8, line 21, change "s" to --a--.

Signed and Sealed this

Twenty-second Day of July, 1997

Bruce Lehman

Attest:

BRUCE LEHMAN

Attesting Officer

Commissioner of Patents and Trademarks

US006625756B1

(12) **United States Patent**
Grochowski et al.

(10) **Patent No.: US 6,625,756 B1**
(45) **Date of Patent: Sep. 23, 2003**

(54) **REPLAY MECHANISM FOR SOFT ERROR RECOVERY**

(75) Inventors: **Edward T. Grochowski**, San Jose, CA (US); **William Rash**, Saratoga, CA (US); **Nhon Quach**, San Jose, CA (US)

(73) Assignee: **Intel Corporation**, Santa Clara, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/469,961**

(22) Filed: **Dec. 21, 1999**

Related U.S. Application Data

(63) Continuation-in-part of application No. 08/994,503, filed on Dec. 19, 1997, now Pat. No. 6,047,370.

(51) **Int. Cl.**[7] **G06F 11/00**
(52) **U.S. Cl.** **714/17**; 714/10
(58) **Field of Search** 714/17, 16, 11, 714/12, 10; 712/230, 219

(56) **References Cited**

U.S. PATENT DOCUMENTS

4,453,215 A		6/1984	Reid	
4,912,707 A	*	3/1990	Kogge et al.	712/228
5,012,403 A		4/1991	Keller et al.	
5,247,628 A	*	9/1993	Grohoski	710/260
5,321,698 A	*	6/1994	Nguyen et al.	714/17
5,475,856 A		12/1995	Kogge	
5,504,859 A	*	4/1996	Gustafson et al.	714/11
5,530,802 A	*	6/1996	Fuchs et al.	714/17
5,530,804 A		6/1996	Edgington et al.	
5,535,410 A		7/1996	Watanabe et al.	
5,561,775 A		10/1996	Kurosawa et al.	
5,604,753 A	*	2/1997	Bauer et al.	714/17
5,630,047 A	*	5/1997	Wang	714/15
5,659,721 A	*	8/1997	Shen et al.	712/228
5,664,214 A		9/1997	Taylor et al.	
5,748,873 A		5/1998	Ohguro et al.	
5,751,985 A		5/1998	Shen et al.	
5,764,971 A		6/1998	Shang et al.	

(List continued on next page.)

FOREIGN PATENT DOCUMENTS

EP	0 315 303 A2	5/1989
EP	0 411 805 A3	2/1991

OTHER PUBLICATIONS

Hennessy, John L., Patterson, David A., "Computer Organization and Design", 1998, Morgan Kaufmann Publishers, Inc., p. G-8.*

Keith Diefendorff, Microprocessor Report, Nov. 15, 1999, pp. 8, vol. 13, No. 15.

Keith Diefendorff, Power4 Focuses on Memory Bandwidth, Oct. 6, 1999, pp. 11-17.

Anthony Marsala & Basel Kanawati; PowerPC Processors; System Theory 1994; Proceedings of the 26th Southwestern Symposium. p.p. 550-556; Athens, OH USA; ISBN: 0-8186-5320; IEEE Catalog # 94TH0599-1.

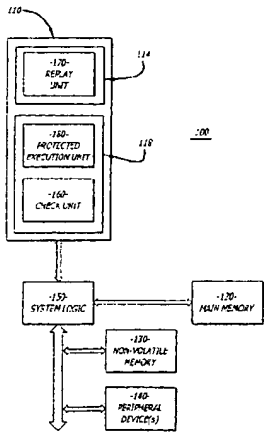
1984 Data Book; ROckwell International, Semiconductor Products Division; Order No. I, Oct. 1983; pp. 3-1-3-22.

Primary Examiner—Robert Beausoliel
Assistant Examiner—Christopher S. McCarthy
(74) *Attorney, Agent, or Firm*—Blakely, Sokoloff, Taylor & Zafman LLP

(57) **ABSTRACT**

A processor is provided that implements a replay mechanism to recover from soft errors. The processor includes a protected execution unit, a check unit to detect errors in results generated by the protected execution unit, and a replay unit to track selected instructions issued to the protected execution unit. When the check unit detects an error, it triggers the replay unit to reissue the selected instructions to the protected execution unit. One embodiment of the replay unit provides an instruction buffer that includes pointers to track issue and retirement status of in-flight instructions. When the check unit indicates an error, the replay unit resets a pointer to reissue the instruction for which the error was detected.

16 Claims, 6 Drawing Sheets



U.S. PATENT DOCUMENTS

5,765,208	A	6/1998	Nelson et al.		
5,784,587	A	7/1998	Lotz et al.		
5,787,474	A	7/1998	Pflum		
5,903,771	A	5/1999	Sgro et al.		
5,966,544	A *	10/1999	Sager	712/1	
6,047,370	A	4/2000	Grochowski		
6,279,119	B1	8/2001	Bissett et al.		
6,393,582	B1	5/2002	Klecka et al.		

* cited by examiner

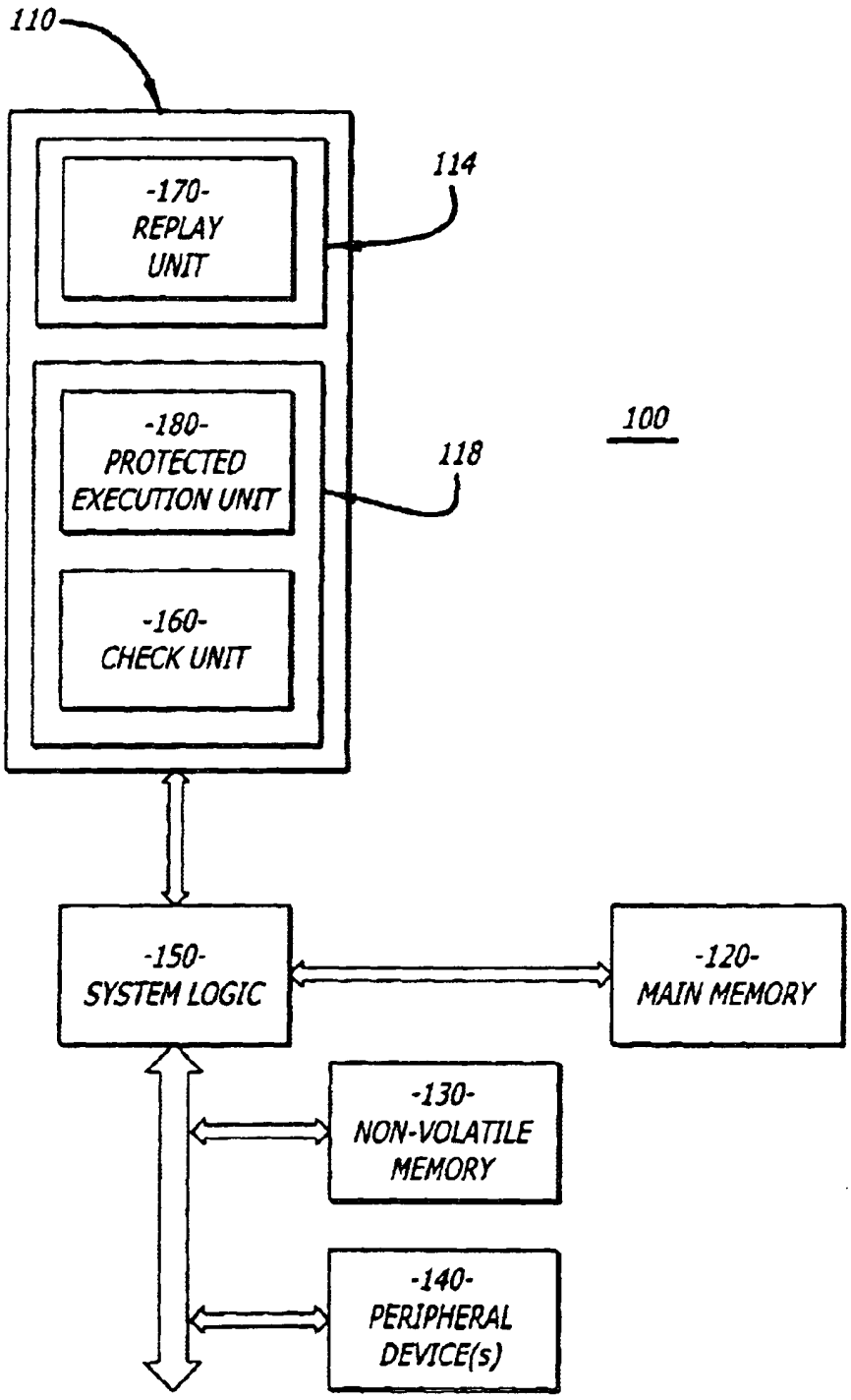

110
-170-
REPLAY
UNIT
114
-180-
PROTECTED
EXECUTION UNIT
118
-160-
CHECK UNIT
100
-150-
SYSTEM LOGIC
-120-
MAIN MEMORY
-130-
NON-VOLATILE
MEMORY
-140-
PERIPHERAL
DEVICE(s)

FIG. 1

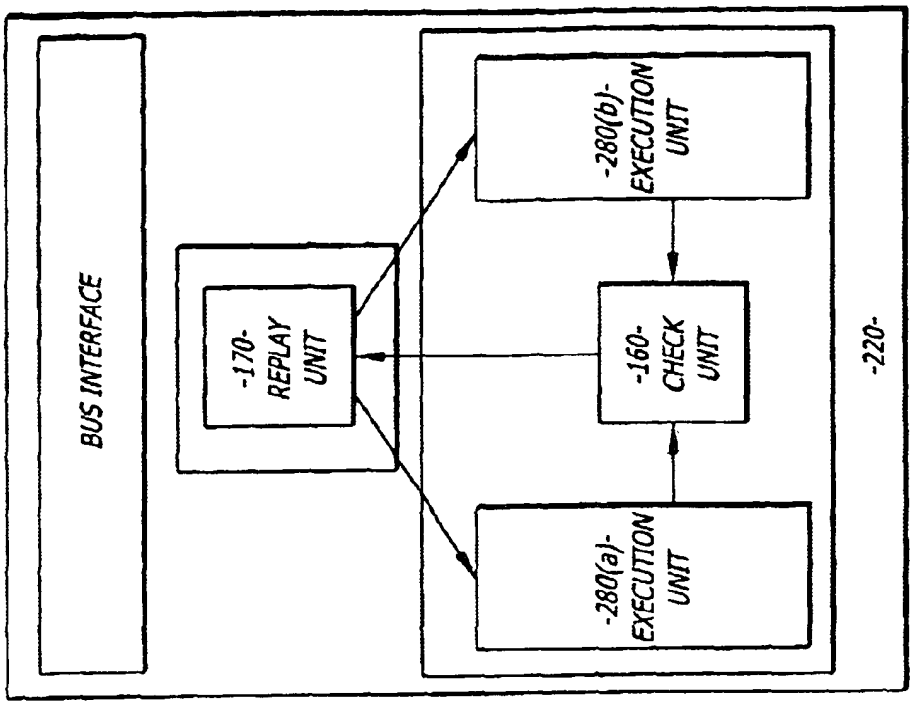

BUS INTERFACE
-170- REPLAY UNIT
-280(b)- EXECUTION UNIT
-160- CHECK UNIT
-280(a)- EXECUTION UNIT
-220-
114
118

FIG. 2B

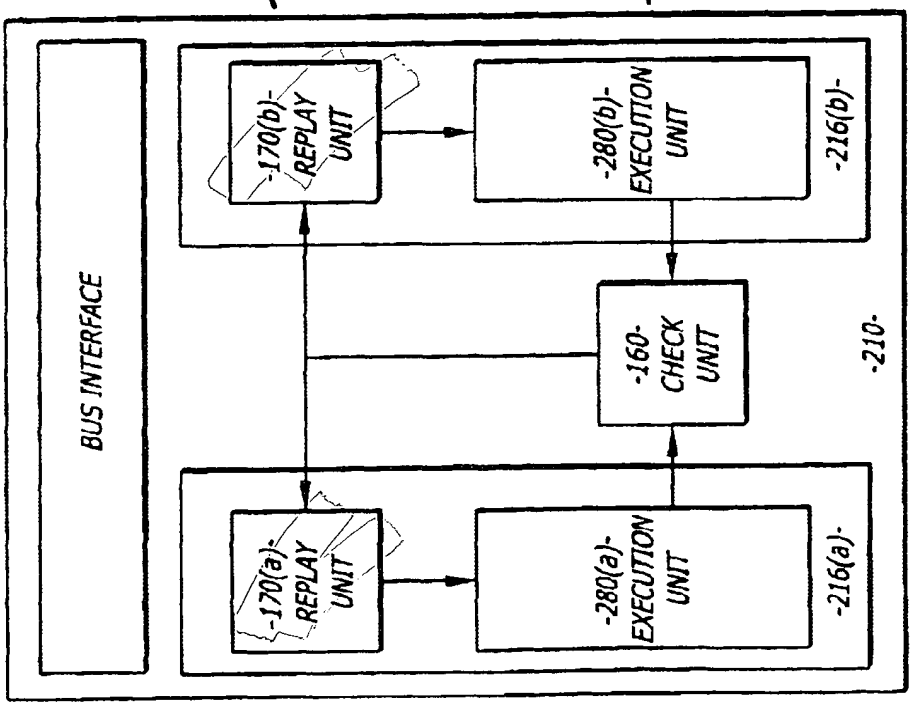

114
118
BUS INTERFACE
-170(b)- REPLAY UNIT
-280(b)- EXECUTION UNIT
-216(b)-
-160- CHECK UNIT
-210-
-170(a)- REPLAY UNIT
-280(a)- EXECUTION UNIT
-216(a)-

FIG. 2A

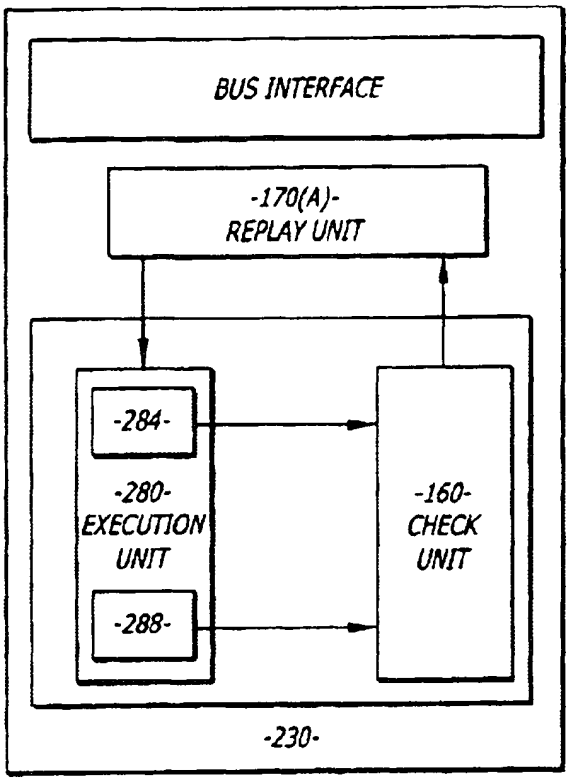

BUS INTERFACE
-170(A)-
REPLAY UNIT
-284-
-280-
EXECUTION
UNIT
-288-
-160-
CHECK
UNIT
-230-

FIG. 2C

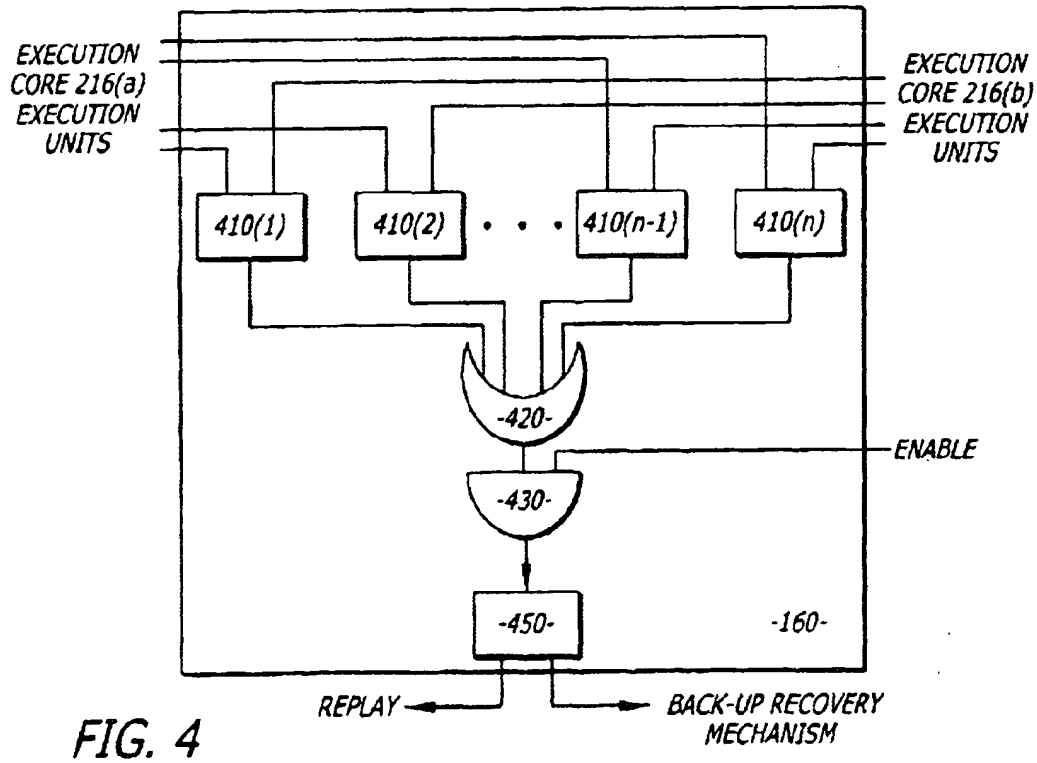

EXECUTION
CORE 216(a)
EXECUTION
UNITS
EXECUTION
CORE 216(b)
EXECUTION
UNITS
410(1)
410(2)
410(n-1)
410(n)
-420-
ENABLE
-430-
-450-
-160-
REPLAY
BACK-UP RECOVERY
MECHANISM

FIG. 4

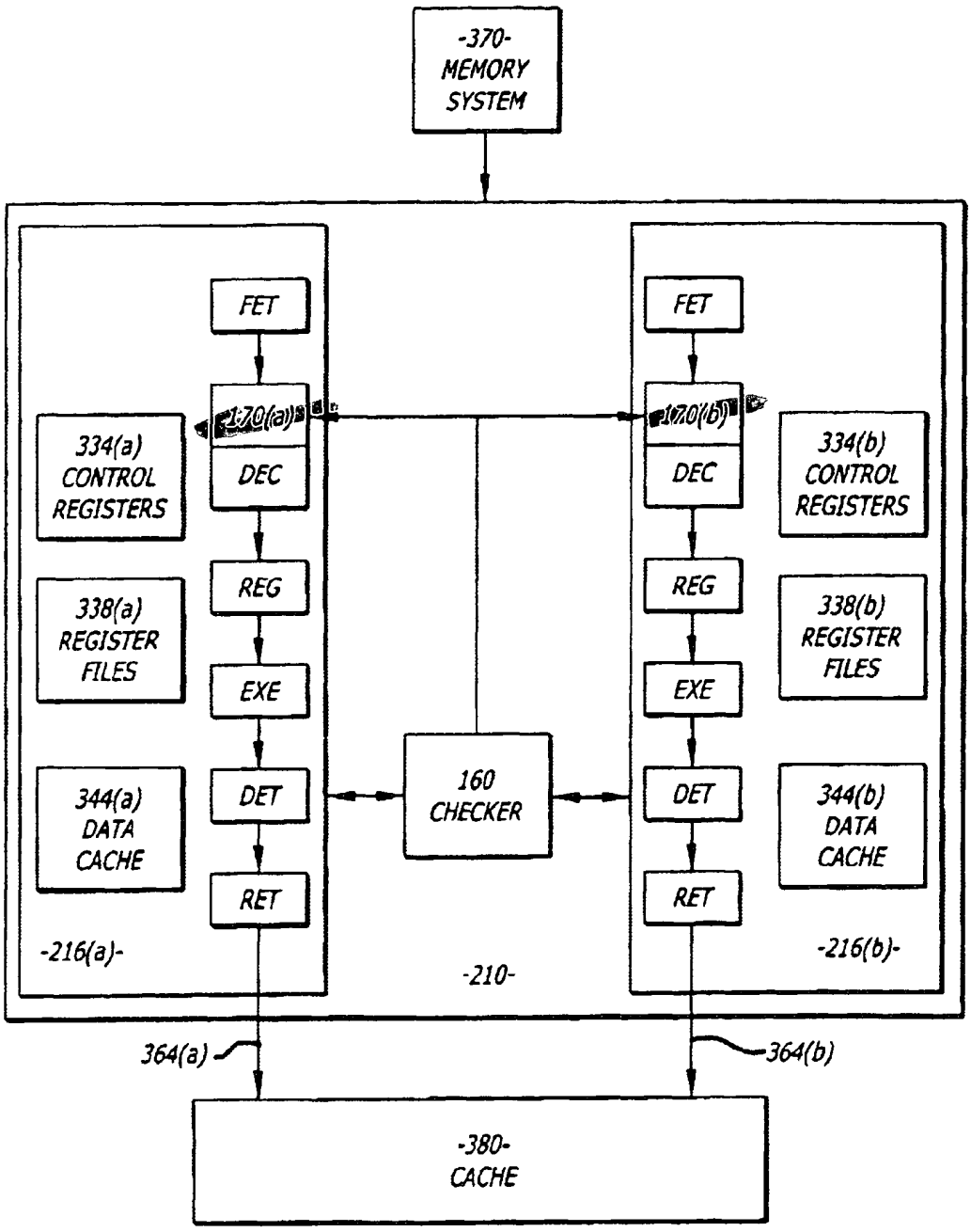

-370-
MEMORY
SYSTEM
FET
170(a)
DEC
REG
EXE
DET
RET
334(a)
CONTROL
REGISTERS
338(a)
REGISTER
FILES
344(a)
DATA
CACHE
-216(a)-
FET
170(b)
DEC
REG
EXE
DET
RET
334(b)
CONTROL
REGISTERS
338(b)
REGISTER
FILES
344(b)
DATA
CACHE
-216(b)-
160
CHECKER
-210-
364(a)
364(b)
-380-
CACHE

FIG. 3

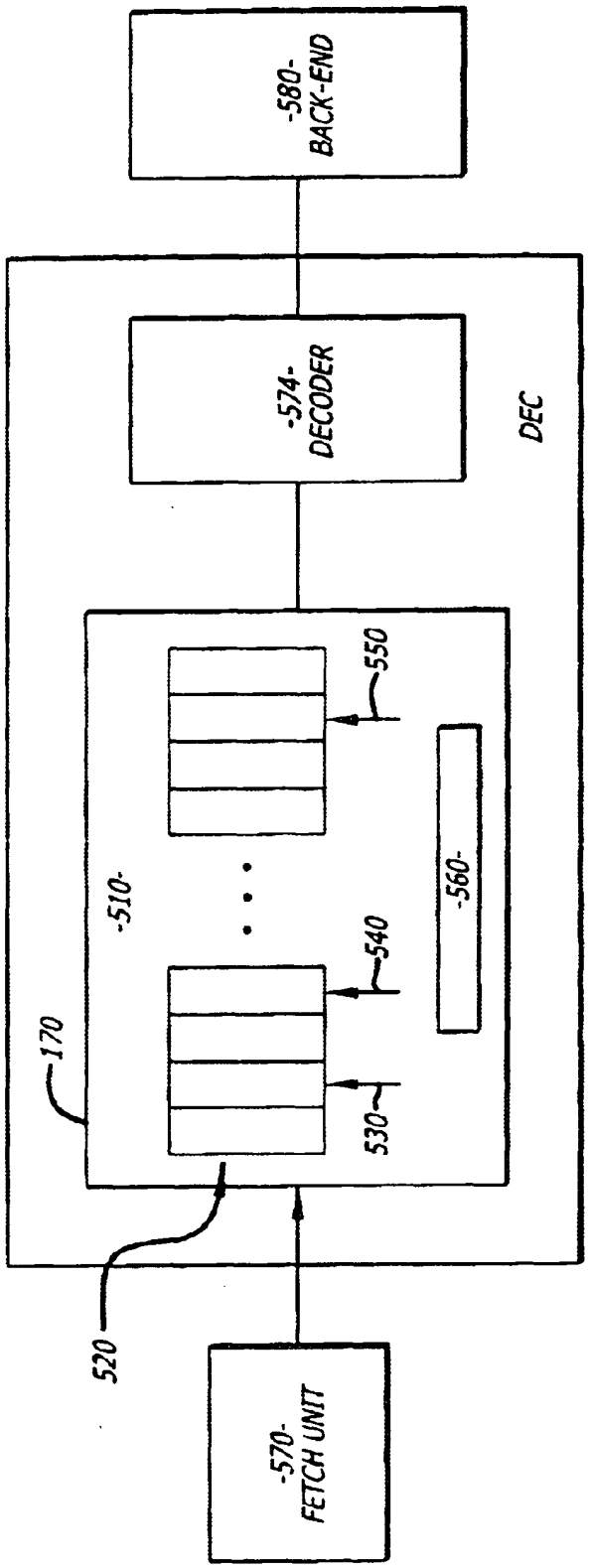

-580-
BACK-END
-574-
DECODER
DEC
550
-510-
-560-
540
170
530
520
-570-
FETCH UNIT

FIG. 5

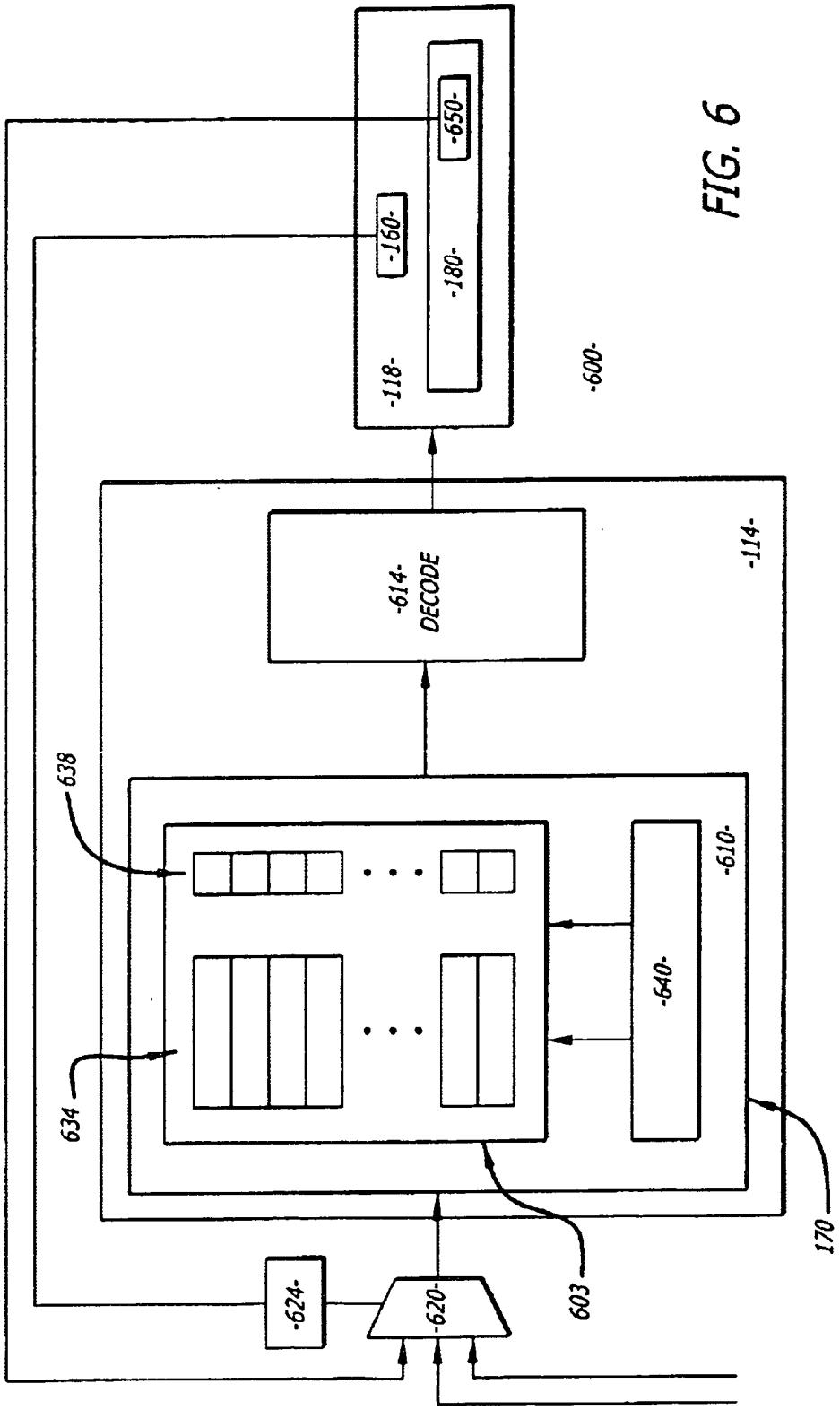

-650-
-160-
-180-
-118-
-600-
-614-
DECODE
-114-
638
-610-
-640-
634
170
603
-624-
-620-

FIG. 6

REPLAY MECHANISM FOR SOFT ERROR RECOVERY

RELATED PATENT APPLICATIONS

This patent application is a continuation-in-part of U.S. patent application Ser. No. 08/994,503, entitled "Processor Pipeline Including Backend Replay", which was filed on Dec. 19, 1997, now U.S. Pat. No. 6,047,370 entitled "CONTROL OF PROCESSOR PIPELINE MOVEMENT THROUGH REPLAY QUEUE AND POINTER BACKUP".

BACKGROUND OF THE INVENTION

1. Technical Field

The present invention relates to microprocessors and, in particular, to microprocessors capable of operating in high-reliability modes.

2. Background Art

Soft errors arise when alpha particles or cosmic rays strike an integrated circuit and alter the charges stored on the voltage nodes of the circuit. If the charge alteration is sufficiently large, a voltage representing one logic state may be changed to a voltage representing a different logic state. For example, a voltage representing a logic true state may be altered to a voltage representing a logic false state, and any data that incorporates the logic state will be corrupted.

Soft error rates (SERs) for integrated circuits, such as microprocessors ("processors"), increase as semiconductor process technologies scale to smaller dimensions and lower operating voltages. Smaller process dimensions allow greater device densities to be achieved on the processor die. This increases the likelihood that an alpha particle or cosmic ray will strike one of the processor's voltage nodes. Lower operating voltages mean that smaller charge disruptions are sufficient to alter the logic state represented by the node voltages. Both trends point to higher SERs in the future. Soft errors may be corrected in a processor if they are detected before any corrupted results are used to update the processor's architectural state.

Processors frequency employ parity-based mechanisms detect data corruption due to soft errors. A parity bit is associated with each block of data when it is stored. The bit is set to one or zero according to whether there is an odd or even number of ones in the data block. When the data block is read out of its storage location, the number of ones in the block is compared with the parity bit. A discrepancy between the values indicates that the data block has been corrupted. Agreement between the values indicates that either no corruption has occurred or two (or four . . .) bits have been altered. Since the latter events have very low probabilities of occurrence, parity provides a reliable indication of whether data corruption has occurred. Error correcting codes (ECCs) are parity-based mechanisms that track additional information for each data block. The additional information allows the corrupted bit(s) to be identified and corrected.

Parity/ECC mechanisms have been applied extensively to caches, memories, and similar data storage arrays. These structures have relatively high densities of data storing nodes and are susceptible to soft errors even at current device dimensions. Their localized array structures make it relatively easy to implement parity/ECC mechanisms. The remaining circuitry on a processor includes data paths, control logic, execution logic and registers ("execution core"). The varied structures of these circuits and their distribution over the processor die make it more difficult to apply parity/ECC mechanisms.

One approach to detecting soft errors in an execution core is to process instructions on duplicate execution cores and compare results determined by each on an instruction by instruction basis ("redundant execution"). For example, one computer system includes two separate processors that may be booted to run in a Functional Redundant Check unit ("FRC") mode. In FRC mode, the processors execute identical code segments and compare their results on an instruction by instruction basis to determine whether an error has occurred. This dual processor approach is costly (in terms of silicon). In addition, inter-processor signaling through which results are compared is too slow to detect corrupted data before it updates the processors' architectural states. Consequently, this approach is not suitable for correcting detected soft errors.

Another computer system provides execution redundancy using dual execution cores on a single processor chip. This approach eliminates the need for inter-processor signaling, and detected soft errors can usually be corrected. However, the processor employs an on-chip microcode to correct soft errors. This approach consumes significant processor area to store the microcode and it is a relatively slow correction mechanism.

The present invention addresses these and other deficiencies of available high reliability computer systems.

SUMMARY OF THE INVENTION

The present invention provides a mechanism for correcting soft errors in high reliability processors.

In accordance with the present invention, a processor includes a protected execution unit, a check unit to detect errors in results generated by the protected execution unit, and a replay unit to track selected instructions issued to the protected execution unit. When the check unit detects an error, it triggers the replay unit to reissue the selected instructions to the protected execution unit.

For one embodiment of the invention, the protected execution unit includes first and second execution units that provide redundant execution results to detect soft errors. For another embodiment of the invention, the protected execution unit includes parity protected storage structures to detect soft errors. For yet another embodiment of the invention, the replay unit provides an instruction buffer that includes pointers to track issue and retirement status of in-flight instructions. When the check unit indicates an error, the replay unit resets a pointer to reissue the instruction for which the error was detected.

BRIED DESCRIPTION OF THE DRAWINGS

The present invention may be understood with reference to the following drawings, in which like elements are indicated by like numbers. These drawings are provided to illustrate selected embodiments of the present invention and are not intended to limit the scope of the invention.

FIG. **1** is a block diagram of a computer system that includes a processor in accordance with the present invention.

FIGS. **2A–2C** are block diagrams of various embodiments of the processor of FIG. **1** representing different types of protected execution units.

FIG. **3** is a more detailed block diagram of one embodiment of the processor shown in FIG. **2A**.

FIG. **4** is a block diagram of one embodiment of the check unit of the processor in FIGS. **2A** and **2B**.

FIG. **5** is a block diagram of one embodiment of a replay unit that may be used to correct soft errors in accordance with the present invention.

FIG. 6 is a block diagram of another embodiment of a replay unit that may be used to correct soft errors in accordance with the present invention.

DETAILED DESCRIPTION OF THE INVENTION

The following discussion sets forth numerous specific details to provide a thorough understanding of the invention. However, those of ordinary skill in the art, having the benefit of this disclosure, will appreciate that the invention may be practiced without these specific details. In addition, various well-known methods, procedures, components, and circuits have not been described in detail in order to focus attention on the features of the present invention.

The present invention provides a hardware mechanism for correcting soft errors in a computer system that is designed to execute instructions with high reliability. High reliability code execution is warranted for certain computer systems that must be available with minimal interruptions in service. When soft errors arise in one of these computer systems as it executes code associated with the operating system kernel or code that operates directly on the platform hardware, the entire computer system can crash. Such errors are to be avoided at all costs. The present invention allows soft errors to be corrected quickly, before the errors have a chance to corrupt architectural data.

Embodiments of the present invention employ a protected execution unit, which processes instructions in a manner that facilitates the detection of soft errors. A check unit monitors the protected execution unit for an indication that a soft error has occurred. The replay unit tracks selected instructions that are in-flight in the protected execution unit. When the check unit indicates that a soft error has occurred, the replay unit reissues the selected in-flight instructions for re-execution.

FIG. **1** is a block diagram of one embodiment of a computer system **100** in which the present invention may be implemented. Computer system **100** includes one or more processors **110**, a main memory **120**, non-volatile memory **130**, various peripheral devices **140**, and system logic **150**. System logic **150** controls data transfers among processor(s) **110**, main memory **120**, non-volatile memory **130**, and peripheral devices **140**. Computer system **100** is provided to illustrate features of the present invention. The particular configuration shown is not necessary to implement the present invention.

Processor **110** implements an instruction execution pipeline comprising a front end **114** and a back end **118**. Front end **114** retrieves instructions and issues them to back end **118** for execution. For the disclosed embodiment of processor **110**, front end **114** includes a replay unit **170**, and back end **118** includes a protected execution unit **180** and a check unit **160**. Front end **114** may retrieve instructions for processing from main memory **120** or non-volatile memory **130**.

Protected execution unit **180** includes circuitry to execute instructions in a manner that facilitates detection of soft errors. This in turn allows code to be executed with high reliability Check unit **160** monitors protected execution unit **180** to determine whether an error has occurred. For one embodiment of processor **110**, protected execution unit **180** implements redundant execution units, and check unit **160** compares results from redundantly executed instructions to determine whether a soft error has occurred. For another embodiment of processor **110**, protected execution unit **180** includes parity-protected storage structures, and check unit **160** monitors data from these structures for parity errors. The present invention does not depend on the particular mechanism through which protected execution unit **180** supports high reliability execution.

Replay unit **170** tracks selected instructions in protected execution unit **180** until they are retired. When an instruction is retired, results generated by the instruction update the architectural state of the processor ("processor state"). For this reason, it is important to detect and correct soft errors before the instructions that are affected by the soft error retire. Since soft errors are a product of transient phenomena (cosmic rays, alpha particles), data corruption attributable to these errors can frequently be eliminated by re-executing instructions that are affected by the soft error. For example, soft errors that corrupt data in execution, control, and delivery circuits are unlikely to recur when instructions are re-executed. These soft errors may be addressed by re-executing selected instructions beginning with the instruction for which the soft error was first detected. Soft errors may also corrupt data in various storage structures associated with the execution resources. Re-executing instructions that merely retrieve corrupted data does not eliminate the problem. However, the corrupted data may be restored by various hardware and software mechanisms, e.g. ECC hardware or firmware. These soft errors may be addressed by re-executing the instructions once the data has been recovered data.

In the following discussion, "instruction" refers to any of the various forms in which instructions are handled by the processor's instruction execution pipeline. These forms include individual instructions or collections of instructions. The latter includes macro-instructions and instruction bundles. For various processor embodiments, instructions or instructions bundles may be decoded into "uops" or instruction syllables, respectively, before they are delivered to the execution units. Where necessary to avoid confusion, the term, "uop", is used to identify the individual components of an instruction that are directed to different types of execution units.

FIG. **2A** is a block diagram of one embodiment of processor **110** (processor **210**) that supports soft error detection through redundant execution clusters. Processor **210** includes a pair of execution cores **216**(*a*), **216**(*b*) (generically, execution core **216**), which are operated in lock step. Each execution core **216** includes a replay unit **170** (**170**(*a*) and **170**(*b*)) and an execution unit **280** (**280**(*a*) and **280**(*b*)). Identical instructions are provided to replay unit **170** by, e.g. a fetch unit (not shown). Each replay unit **170** directs the instruction to its associated execution unit **280** and monitors the issued instructions until they retire.

Results generated by execution units **280** are compared by check unit **160** and a discrepancy indicates a soft error may have occurred. When a discrepancy is detected, check unit **160** signals an error to replay unit **170**, which reissues selected instructions. If the soft error was transient, e.g. a bit flipped in a logic or control circuit, the discrepancy disappears when the instructions are re-executed. If the discrepancy is not eliminated by re-execution, processor **210** may invoke a back-up recovery mechanism. The discrepancy may persist, for example, if data in a register file or data cache of processor **210** was corrupted by a soft error. For one embodiment of processor **210**, check unit **160** invokes a firmware error recovery routine in non-volatile memory **130** if re-executing instructions a selected number of times fails to eliminate the discrepancy.

FIG. **2B** represents another embodiment of processor **110** (processor **220**) that supports soft error detection through

redundant execution. For the disclosed embodiment of processor **220** only duplicates portions of the processor hardware in back end **118**. Protected execution unit **180** includes first and second execution units **280**(*a*) and **280**(*b*). A single replay unit **170** provides identical instructions to execution units **280** and tracks them until they retire. As for the case of processor **210**, processor **220** provides a level of redundant execution that allows soft errors to be detected more easily. However, only the back end stages of processor **220** are duplicated. This reduces the hardware cost for processor **220**, but processor **220** may be more susceptible to soft errors in front end **114**. As in the embodiment of FIG. **2**A, check unit **160** monitors execution units **280** and signals replay unit **170** when a discrepancy is detected. Processor **220** may also implement a back-up recovery mechanism for those cases in which re-execution does not eliminate the discrepancy.

FIG. **2**C is a block diagram representing yet another embodiment of processor **110**(processor **230**). Processor **230** supports soft error detection through parity protected storage structures. For the disclosed embodiment of processor **230**, protected execution unit **180** comprises execution unit **280** having a parity-protected register file **284** and cache **288**. In addition, various latches and other storage structures in the control and execution circuitry of execution unit **280** may incorporate parity protection. Check unit **160** monitors the parity protected storage structures and signals when a parity error is detected. For example, the parity of data blocks in, e.g., register file **284** or cache **288** is indicated through a corresponding parity bit. When a data block is accessed, the parity bit may be compared with a parity value calculated for the data block. A discrepancy between the stored and calculated parity values indicates a soft error corrupted the data after it was stored. For one embodiment of processor **230**, check unit **160** includes hardware to implement the parity check.

FIGS. **2**A–**2**C represent a sample of the different types of protection a processor may employ to support high reliability execution. Protected execution unit **180** may employ other mechanisms to support high reliability execution. In the following discussion, the present invention is illustrated in greater detail, using an embodiment of processor **210**. Persons skilled in the art of processor design will appreciate the modifications necessary to implement the replay unit **170** for other embodiments of processor **110**.

FIG. **3** represents in greater detail one embodiment of processor **210**. For the disclosed embodiment, each execution core **216** is represented as a series of stages in an instruction execution pipeline. Each stage corresponds to one or more operations implemented by execution cores **216** to execute their instructions. Alternatively, the pipeline stages may be understood to represent the logic that executes the indicated operations. Instructions and data are provided to execution cores **216** from a memory system **370**. Memory system **370** may represent, for example, main memory **120** and non-volatile memory **130** of FIG. **1**. Cache **380** represents a portion of memory system **370** to which results from executed instructions are written. Cache **380** may be located on the same chip as processor **210** or it may be located on a separate chip.

For the disclosed embodiment of processor **210**, each execution core **216** is partitioned into a fetch (FET) stage, a decode (DEC) stage, a register (REG) stage, an execute (EXE) stage, a detect (DET) stage, and a retirement (RET) stage. One or more instructions are retrieved from memory system **370** in FET stage. The retrieved instructions are decoded into μops in DEC stage, and source operands specified by the μop(s) are retrieved in REG stage. The μop(s) are executed on the retrieved operands in EXE stage, and any exceptions raised by the μop(s) are signaled in DET stage. The μop(s) is retired in RET stage if no exceptions are detected.

For the disclosed embodiment, results from retired μop(s) are written to cache **380** through retirement channel **364**. Because execution cores **216**(*a*), **216**(*b*) operate redundantly, only one of retirement channels **364** needs to update cache **380**. One embodiment of processor **210** may implement a high performance mode in which execution cores **216** operate independently. For this embodiment, both retirement channels **364** are active.

For the disclosed embodiment of processor **210**, replay unit **170** is represented as part of DEC stage, although it may be incorporated in FET stage for other embodiments (FIG. **6**). Replay unit **170** include a buffer (FIG. **5**) to temporarily stores fetched instructions and control logic (FIG. **5**) to adjust one or more pointers to indicate the status of the stored instructions. Incorporating replay unit **170** in the instruction execution pipelines of processor **210** allows a fast response to any error induction provided by check unit **160**. In addition, the buffer of replay unit **170** serves the additional function of decoupling back end stages (REG, EXE, DET, RET) from front end stages (FET, DEC). This allows front-end operations to continue, even if back-end operations are stalled or otherwise delayed. It also allows back-end operations to proceed if front-end operations are delayed.

The present invention does not require partition of processor **100** into a particular set of pipeline stages. For example, a disclosed stage may be subdivided into two or more stages to address timing issues or facilitate higher processor clock speeds. Alternatively, two or more stages may be combined into a single stage. Other embodiments may include hardware for processing instructions out-of-order. The disclosed pipeline provides only one example of how operations may be partitioned in a processor implementing the present invention.

Also shown for each execution core **216** are status/control (S/C) registers **334**, data registers **338**, and a data cache **344**. S/C registers **334** store information that governs the operation of execution core **216**. Data registers **338** store operands for use by various resources in execution core **110**, and data cache **344** buffers operands between memory system **370** and other resources in execution core **216**. Depending on timing constraints, data cache **344** may provide operands to data registers **338** or directly to execution resources in EXE stage **340**.

Execution cores **216**(*a*) and **216**(*b*) are synchronized to operate on identical instructions in lock step to support high reliability execution. One embodiment of processor **210** may provide a high performance (HP) mode in addition to the high reliability (HR) mode. In HP mode, execution cores **216**(*a*) and **216**(*b*) operate on different instructions. For example, processor **210** may operate as a single chip symmetric multi-processing (SMP) system in HP mode, with each execution core **216** operating as an independent processor core. Dual mode embodiments of processor are described in U.S. patent application Ser. No. 09/470,096, entitled "Microprocessor Having a High Reliability Operating Mode" and filed on even date herewith, and U.S. patent application Ser. No. 09/470.098, entitled "Microprocessor Having a High Reliability Operating Mode" and filed on even date herewith.

FIG. **4** is a block diagram representing one embodiment of check unit **160** that is suitable for use with processors **210**,

220. The disclosed embodiment of check unit **160** includes "n" comparators **410(1)**–**410**(*n*), an OR gate **420**, and an AND gate **430**. A comparator **410** is provided for each execution unit in execution core **216** (FIG. **3**). For example, one embodiment of processor **210** may include an integer execution unit (IEU), a floating point execution unit (FPU), a memory execution unit (MEU), and a branch execution unit BRU in the EXE stage of each execution core **216** (FIG. **3**). For this embodiment, check unit **160** includes 4 comparators **410**. Comparator **410(1)**, **410(2)**, **410(3)** and **410(4)** monitor outputs of the IEUs, FPUs, MEUs, and BRUs, respectively, from execution cores **216**(*a*), **216**(*b*).

For the disclosed embodiment of check unit **160**, each comparator **410** generates a logic value zero when the execution results applied to its inputs match and a logic value one when the execution results do not match. For one embodiment of check unit **160**, comparators **410** are self check comparators. OR gate **420** generates a logic value one when any of comparators **410** indicates that its corresponding execution results do not match. The output of OR gate **20** indicates an error when AND gate **430** is enabled. This error signal may be used to trigger a flush of the processor's instruction execution pipeline and a re-execution of the appropriate instructions by replay unit. Pipeline flushing operations may be handled through an exception handling unit in the processor (not shown). Mechanisms for flushing processor pipelines are well-known.

For another embodiment of the invention, replay may be initiated in the FET stage of the processor's instruction execution pipeline. For example, when check unit **160** detects an error, an instruction pointer (IP) associated with the instruction currently in the DET stage may be provided to the FET stage. The instructions to be re-executed may then be retrieved from an instruction cache associated with the FET stage. The exception handling unit may provide the resteer address to the FET stage. This embodiment is discussed in greater detail in conjunction with FIG. **6**.

As discussed above, one embodiment of processor **210** may be switched between a high reliability (HR) mode, in which execution cores **216** operate in lock step, and a high performance (HP) mode, in which execution cores **216** operate on different instruction segments. The ENABLE input to AND gate **430** allows check unit **160** to be disabled when processor **210** is in HP mode.

Embodiments of check unit **160** may include a counter **450** to track the number of replays triggered on a particular instruction. For example, an embodiment of processor **110** that employs a back-up recovery mechanism may invoke the back-up recovery mechanism after a specified number of re-execution attempts fail to eliminate a discrepancy. For these embodiments, counter **450** may track the replay attempts and invoke a recovery routine when the specified number is reached.

Persons skilled in the art of processor design and having the benefit of this disclosure will recognize other variations on check unit **160** that may be activated to monitor results in execution cores **216**.

For the disclosed embodiments of processor **110**, check unit **160** compares execution results in the DET stage, to determine whether an error has occurred. When no discrepancy is detected, the corresponding instruction(s) are allowed to retire. The recovery mechanism is implemented when a discrepancy or mismatch between execution results is detected.

The soft errors targeted by the present invention are unlikely to occur in both execution cores simultaneously. Consequently, differences in execution results detected by check unit **160**, in the absence of errors originating in parity/ECC protected arrays, are most likely due to soft errors in the circuitry of execution cores **216**. Since these errors occur relatively infrequently, they may be corrected by flushing "in-flight" instructions from the execution cores/clusters (or portions thereof) and re-executing the flushed instructions, beginning with the instruction that triggered the error.

For one embodiment of the invention, the replay unit tracks each instruction until it is successfully retired. If an error (mismatch between execution results) is detected for the instruction in DET stage, each execution core or a portion of it may be resteered to reexecute selected instructions, beginning with the instruction currently in DET stage.

FIG. **5** is a block diagram of one embodiment of replay unit **170** and associated logic. The disclosed embodiment of replay unit **170** includes multiple slots **520** to store fetched instructions, pointers **530**, **540**, **550** to track the status of the stored instructions, and control logic **560** to manage pointers **530**, **540**, **550**. For the disclosed embodiment, a fetch unit **570** provides an instruction (or instruction bundle) to an available slot **520**. The stored instruction(s) is decoded into one or more μops by a decoder **574** and issued to a back end **580** of the processor pipeline. Back end **580** may include, for example, circuitry associated with the REG, EXE, DET, and RET stages of execution cores **216**.

For an alternate embodiment of replay unit **170**, decoder **574** may operate on instructions before they are stored in slots **520**. For yet another embodiment, fetch unit **570** may provide instruction bundles to replay unit **170**, which are then mapped to specific execution units by decoder **574**. The extent of DEC stage for the embodiment of processor **210** is indicated in the figure.

Control unit **560** updates pointers **530**, **540**, **550** as new μops are transferred to queue **510**, issued to back-end **580**, and retired, respectively. For other embodiments, the relative location of replay unit **170** and decoder **514** may be reversed, in which case replay unit **170** stores μops decoded from fetched instructions. In the following discussion, "instruction" and "μop" are used interchangeably.

For the disclosed embodiment of processor **210**, replay unit **170** may be incorporated in the logic associated with DEC stage (FIG. **4**) and back-end **580** includes logic associated with REG, EXE, DET, and RET stages. Pointers **530**, **540**, **550** are updated as instructions are received from FET stage, transferred to REG stage, and retired in RET stage, respectively. For this embodiment, pointer **530** ("head pointer") indicates the latest instruction(s) to enter queue **510**, pointer **540** ("tail pointer") indicates the next instruction(s) to be issued to the REG stage, and pointer **550** indicates the next instruction to be retired ("replay pointer") from RET stage. At a given time, the instructions in the slots that follow tail pointer **540**, up to and including the instruction(s) indicated by replay pointer **550**, are being executed ("in-flight") in back-end **580**. Head pointer **530** is updated when a new instruction enters REG stage, tail pointer **540** is updated when a new instruction enters replay unit **170** from instruction cache **570**, and replay pointer **550** is updated when the instruction to which it currently points enters RET stage

When the disclosed embodiment of processor **110** is operating in redundant mode, check unit **160** signals an error and flushes the back end pipe stages if it detects discrepancy between the execution results in the DET stages of execution

cores 216(*a*) and 216(*b*). When control unit **560** detects the error signal, it adjusts tail pointer **530** to indicate the slot currently indicated by replay pointer **550**. This effectively reschedules all un-retired instructions that are currently in the back end of the pipeline for (re)issue to the REG stage. For one execution core/cluster, the instruction(s) indicated by replay pointer **550** is the source of the erroneous execution result, and the instruction(s) in the slots between head pointer **530** and replay pointer **550** follow this error-generating instruction in the back-end of the pipeline. All of these instruction(s) may be flushed from the back end of the pipeline, and reissued by replay unit **170**, beginning with the instruction(s) that triggered the error.

Another embodiment of replay unit **170** tracks dependencies between instructions in addition to their status in backend **580**. This embodiment of replay unit **170** flushes and replays only the instructions that triggered the error and the issued μops that depend on it ("partial replay").

Yet another embodiment of replay unit **170** employs a shift register, which physically shifts instruction(s) down the queue **510** as earlier instruction(s) are retired. In this embodiment, the oldest, unretired instruction(s) is at the end of queue **510**, and a separate replay pointer is not needed. As in the above embodiments, head pointer **530** indicates the next instruction to issue to the back end and tail pointer **540** indicates the last instruction to enter queue **510**.

FIG. **6** is a block diagram of an embodiment of a processor **600** in which re-execution is controlled through a fetch unit **610** of a processor's instructions execution pipeline. For the disclosed embodiment, front end **114** includes fetch unit **610** and decode unit **614**, and back end **118** includes protected execution unit **180** and check unit **160**. In addition, an exception handling unit **650** is shown as part of protected execution unit **180**.

An IP selection MUX **620** receives IPs from various sources in the instruction execution pipeline. An associated selection logic **624** receives control signals from various sources in the pipeline, prioritizes them, and selects an IP to forward to fetch unit **610** according to the prioritized signals. One of the signals provided to selection logic **624** is a resteer signal from check unit **160**. Other components may be provided by, e.g., branch execution units, exception unit **650**, and various other components of processor **600** that can alter the instruction flow through the pipeline. Each IP provided at the output of MUX **620** may point to a single instruction or a bundle of instructions, depending on the particular embodiment of processor **600**.

For the disclosed embodiment, fetch unit **610** includes an instruction cache **630** and control logic **640**. Instruction cache **630** includes instruction entries **634** to store instructions for processing and status entries **638** to indicate the status of the various instructions. For the present invention, status entries indicate when a corresponding instruction in one of entries **634** may be evicted from cache **630**. For one embodiment of the present invention, control logic **640** receives an indication from exception unit **650** when an instruction retires, and indicates in the appropriate status entry **638** that the instruction in the corresponding entry **634** may be replaced. Control logic **640** may employ various criteria for replacing instructions in cache **630** in addition to whether the instruction has retired. For a preferred embodiment of fetch unit **610**, no instruction is considered available for retirement until it has been retired.

When MUX **620** selects an IP for processing, control logic **640** reads the selected instruction out of an appropriate entry **634** in cache **630** and forwards it to decode unit **614**. Decode unit **614** issues the instruction to an appropriate execution unit in protected execution unit **180**. Check unit **160** monitors protected execution unit **180** for any errors. If an error is detected, exception unit **650** indicates a resteer IP to MUX **620** and check unit **160** triggers MUX **620** to select the IP provided by exception unit **650**. In this way, the instruction corresponding to the resteer IP and the instructions that follow it in execution order are run through the instruction execution pipeline again.

Replay unit **170** provides a relatively efficient hardware mechanism for correcting soft errors associated with logic, latches, and other storage locations in execution cores **216**. It eliminates the need for providing parity protection for these locations. As noted above, soft errors in certain storage resources can not be corrected by replay unit **170**. For example, when a soft error corrupts an operand in one of the data register files, re-executing instructions on the corrupted input data will not alleviate the mismatch between instruction results generated with the corrupted and uncorrupted data. For these and similar errors that can not be corrected through replay, a fall back error correction mechanism may be provided.

One approach to these errors is to provide ECC protection for the storage structures. This is typically done for certain caches, and it allows parity errors to be detected or corrected on the fly. Corrupted data is detected and corrected before it generates mismatches in execution results. Providing ECC protection for all such storage structures is very costly in terms of silicon die area. Another approach is to provide parity protection. This allows corrupted data to be identified relatively quickly. Since these errors are not corrected through replay, another mechanism is provided for this purpose.

For one embodiment of computer system **100**, a recovery routine is provided through non-volatile memory **140**. Check unit **160** may trigger a machine check that invokes a firmware-based error handling routine. For this embodiment, processor **110** may access an error handling routine when check unit **160** signals an error and replay unit **170** fails to correct it after a specified number of tries. One embodiment of a firmware recovery mechanism operates in conjunction with parity protected storage locations. When replay fails to correct a mismatch in execution results, a recovery routine is implemented to read parity bits associated with the storage structures to locate the error. The storage location that produces the error may be updated with data from the execution core that does not display any parity errors. A firmware based mechanism for processing soft errors is described in U.S. patent application Ser. No. 09/469,963, entitled "Firmware Mechanism for Correcting Soft Errors" and filed on even date herewith.

There has thus been provided processor including a hardware-based mechanism for correcting soft errors. The processor includes a protected execution unit, a check unit, and a replay unit. The protected execution unit is designed to facilitate detection of soft errors. The check unit monitors the protected execution unit for indications of soft errors and signal the replay unit when an error is indicated. The replay unit issues instructions to the protected execution unit and temporarily stores an indication of the issued instructions while they are in-flight. If an error is indicated, the replay unit reissues selected instructions for reexecution.

For soft errors in longer term storage structures, e.g. register files, low level caches and the like, corrupted data is only by regenerating the original data. This may be done through ECC mechanisms or through a separate error recov-

ery routine. These errors may be identified by replaying instructions and rechecking the instruction results for mismatch. If the mismatch persist through replay, it is likely attributable to corrupted data, and a firmware recovery mechanism may be implemented.

The disclosed embodiments have been provided to illustrate various features of the present invention. Persons skilled in the art of processor design, having the benefit of this disclosure, will recognize variations and modifications of the disclosed embodiments, which none the less fall within the spirit and scope of the appended claims.

We claim:

1. A processor comprising:

a protected execution unit to process instructions;

a check unit to detect an error associated with processed instructions; and

a replay queue to issue instructions to the protected execution unit for processing, to track the issued instructions, and to reissue selected issued instructions when the check unit detects an error, the replay queue including first and second pointers to indicate a next instruction to issue and a next instruction to retire, wherein the replay queue adjusts the first and second pointers to reissue instructions to the execution unit beginning with an instruction that generated a result mismatch.

2. The processor of claim 1, wherein

the protected execution unit comprises first and second execution units to process instructions in lock step and the replay queue comprises first and second replay queues to provide instructions to the first and second execution units, respectively.

3. The processor of claim 1, wherein

instructions are flushed from the execution unit when the check unit indicates an error.

4. The processor of claim 1, wherein

the execution units operate in lock step when the processor is in a high reliability mode and

the execution units operate independently when the processor is in a high performance mode.

5. The processor of claim 1, wherein

the processor implements a recovery algorithm if an instruction that triggers a replay generates a mismatch when it is replayed.

6. A method for executing instructions with high reliability, comprising:

storing an instruction temporarily in a replay buffer;

issuing the instruction to a protected execution unit including

staging the instruction to the protected execution unit, and

adjusting a first flag in the buffer to indicate the instruction has been issued including setting a first pointer to indicate a buffer slot in which the issued instruction is stored;

setting a second pointer to indicate a buffer slot in which a next instruction to retire is stored;

checking results generated by the instruction in the protected execution unit; and

reissuing the instruction to the protected execution unit if an error is indicated, wherein reissuing the instruction includes copying the second flag to the first flag.

7. The method of claim 6, further comprising:

retiring the instruction when no error is indicated.

8. The method of claim 7, wherein

retiring the instruction includes

adjusting a second pointer to indicate the instruction has retired; and

updating an architectural state data with the result generated by the instruction.

9. A computer system comprising:

a processor including

a protected execution unit to execute instructions in a manner that facilitates soft error detection,

a check unit to monitor the protected execution unit and to generate a signal when an error is indicated,

a replay unit to provide instructions to the protected execution unit, to track the instructions until they are retired, and to replay selected instructions when the check unit indicates an error, the replay unit includes first and second pointers to indicate a next instruction to issue and a next instruction to retire, respectively, and

a storage structure to provide a recovery algorithm to the processor when replay of selected instructions does not eliminate the mismatch;

wherein the protected execution unit is flushed prior to the replay when an error is indicated; and

wherein the replay unit and the protected execution unit are flushed prior to implementing a recovery routine.

10. The computer system of claim 9, wherein

the storage structure is a non-volatile memory structure.

11. The computer system of claim 9, wherein

the protected execution unit comprises first and second execution units and the replay unit provides identical instructions to the first and second execution units.

12. The computer system of claim 9, wherein

the protected execution unit to execute instructions to facilitate detection of soft errors.

13. The computer system of claim 12, wherein

the protected execution unit includes redundant execution units that execute instructions to facilitate detection of soft errors.

14. The computer system of claim 12, wherein

the protected execution unit includes parity-protected storage structures to execute instructions to facilitate detection of soft errors.

15. A processor comprising:

first and second execution cores to process identical instructions in lock step, each execution core including a replay unit to track instructions that have yet to retire, each replay unit including

buffer slots to store instructions for execution and

a first and second pointers to indicate a next instruction to issue and a next instruction to retire, respectively;

a check unit to compare instructions results generated by the execution cores and to trigger the replay unit to resteer the first and second execution cores to an instruction when the instruction results generate a mismatch; and

wherein each replay unit copies the second pointer to the first pointer when the instruction results generate a mismatch.

16. The processor of claim 15, wherein

the check unit signals an instruction flush when a mismatch is detected.

* * * * *

US006502185B1

(12) United States Patent
Keller et al.

(10) Patent No.: US 6,502,185 B1
(45) Date of Patent: Dec. 31, 2002

(54) PIPELINE ELEMENTS WHICH VERIFY PREDECODE INFORMATION

(75) Inventors: **James B. Keller**, Palo Alto, CA (US); **Puneet Sharma**, Singapore (SG); **Keith R. Schakel**, San Jose, CA (US); **Francis M. Matus**, Sunnyvale, CA (US)

(73) Assignee: **Advanced Micro Devices, Inc.**, Sunnyvale, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/476,936**

(22) Filed: **Jan. 3, 2000**

(51) **Int. Cl.**[7] **G06F 9/30**
(52) **U.S. Cl.** **712/213**
(58) **Field of Search** 712/213

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,689,672 A	*	11/1997	Witt et al.	711/118
5,748,978 A		5/1998	Narayan et al.	712/23
5,819,059 A	*	10/1998	Tran	712/210
5,850,532 A		12/1998	Narayan et al.	712/213
5,968,163 A		10/1999	Narayan et al.	712/204
5,970,235 A	*	10/1999	Witt et al.	712/213
6,125,441 A	*	9/2000	Green	712/204
6,189,087 B1	*	2/2001	Witt et al.	712/208
6,405,303 B1	*	6/2002	Miller et al.	712/204

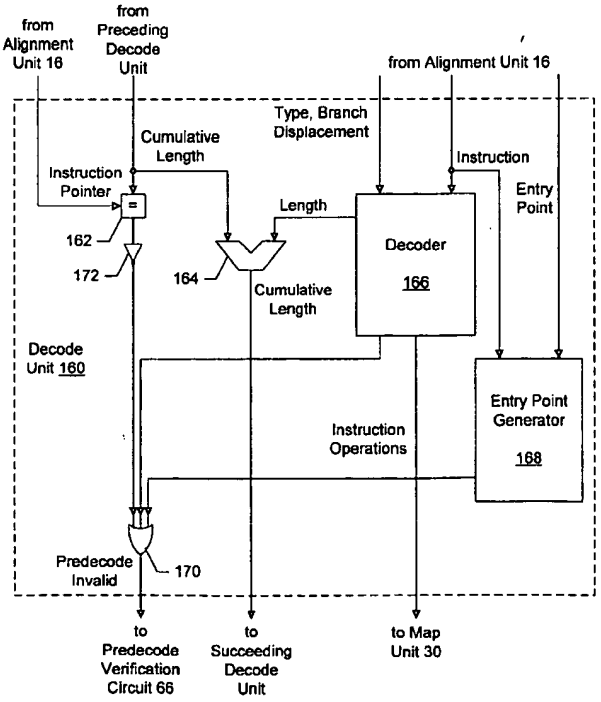
* cited by examiner

Primary Examiner—Eric Coleman
(74) *Attorney, Agent, or Firm*—Lawrence J. Merkel

(57) ABSTRACT

A processor includes an instruction cache and a predecode cache which is not actively maintained coherent with the instruction cache. The processor fetches instruction bytes from the instruction cache and predecode information from the predecode cache. Instructions are provided to a plurality of decode units based on the predecode information, and the decode units decode the instructions and verify that the predecode information corresponds to the instructions. More particularly, each decode unit may verify that a valid instruction was decoded, and that the instruction succeeds a preceding instruction decoded by another decode unit. Additionally, other units involved in the instruction processing pipeline stages prior to decode may verify portions of the predecode information. If the predecode information does not correspond to the fetched instructions, the predecode information may be corrected (either by predecoding the instruction bytes or by updating the predecode information, if the update may be determined without predecoding the instruction bytes). In one particular embodiment, the predecode cache may be a line predictor which stores instruction pointers indexed by a portion of the fetch address. The line predictor may thus experience address aliasing, and predecode information may therefore not correspond to the instruction bytes. However, power may be conserved by not storing and comparing the entire fetch address.

22 Claims, 11 Drawing Sheets



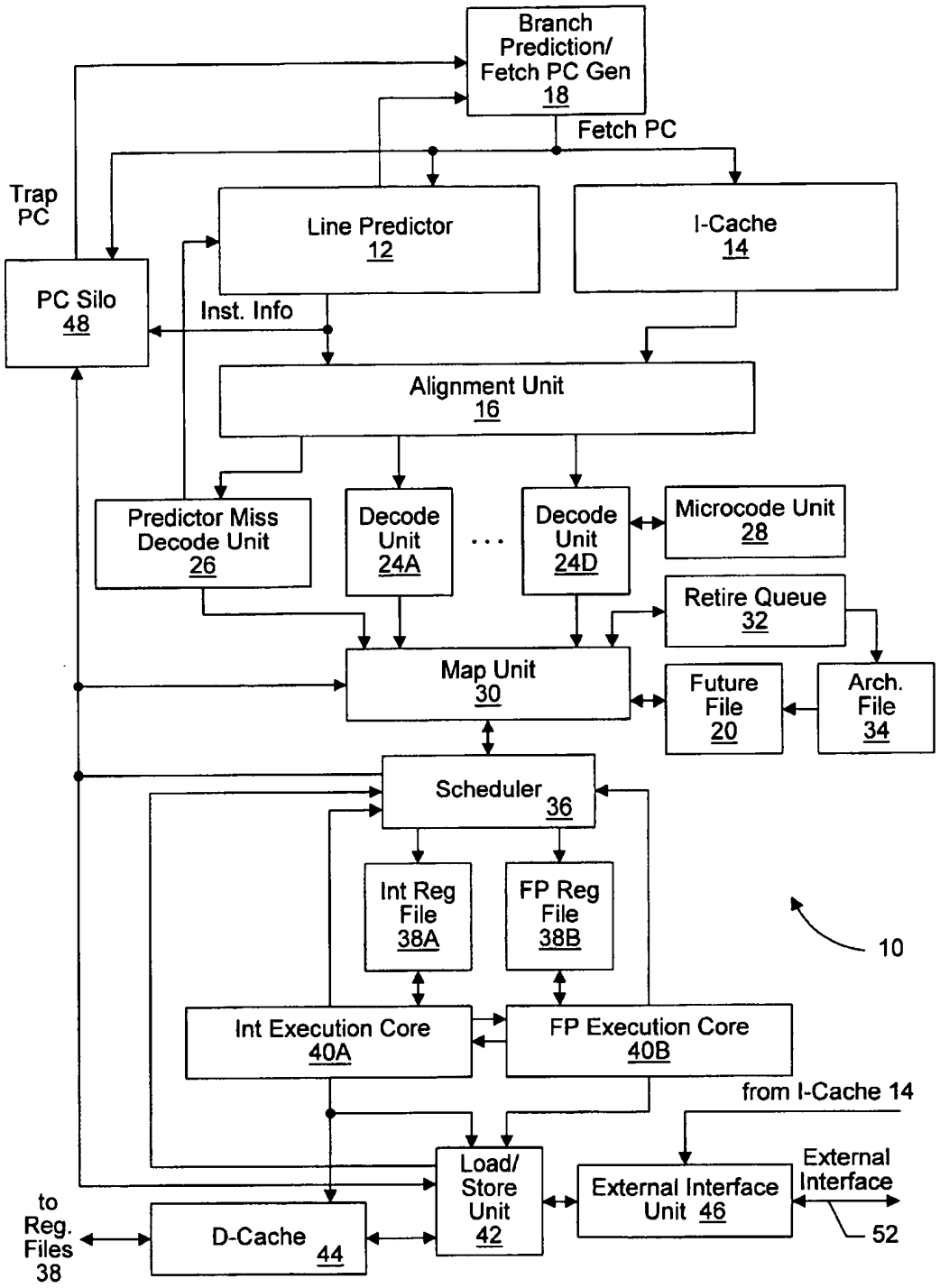

Branch Prediction/ Fetch PC Gen 18
Fetch PC
Trap PC
Line Predictor 12
I-Cache 14
PC Silo 48
Inst. Info
Alignment Unit 16
Predictor Miss Decode Unit 26
Decode Unit 24A
Decode Unit 24D
Microcode Unit 28
Retire Queue 32
Map Unit 30
Future File 20
Arch. File 34
Scheduler 36
Int Reg File 38A
FP Reg File 38B
10
Int Execution Core 40A
FP Execution Core 40B
from I-Cache 14
Load/ Store Unit 42
External Interface Unit 46
External Interface
52
to Reg. Files 38
D-Cache 44

Fig. 1

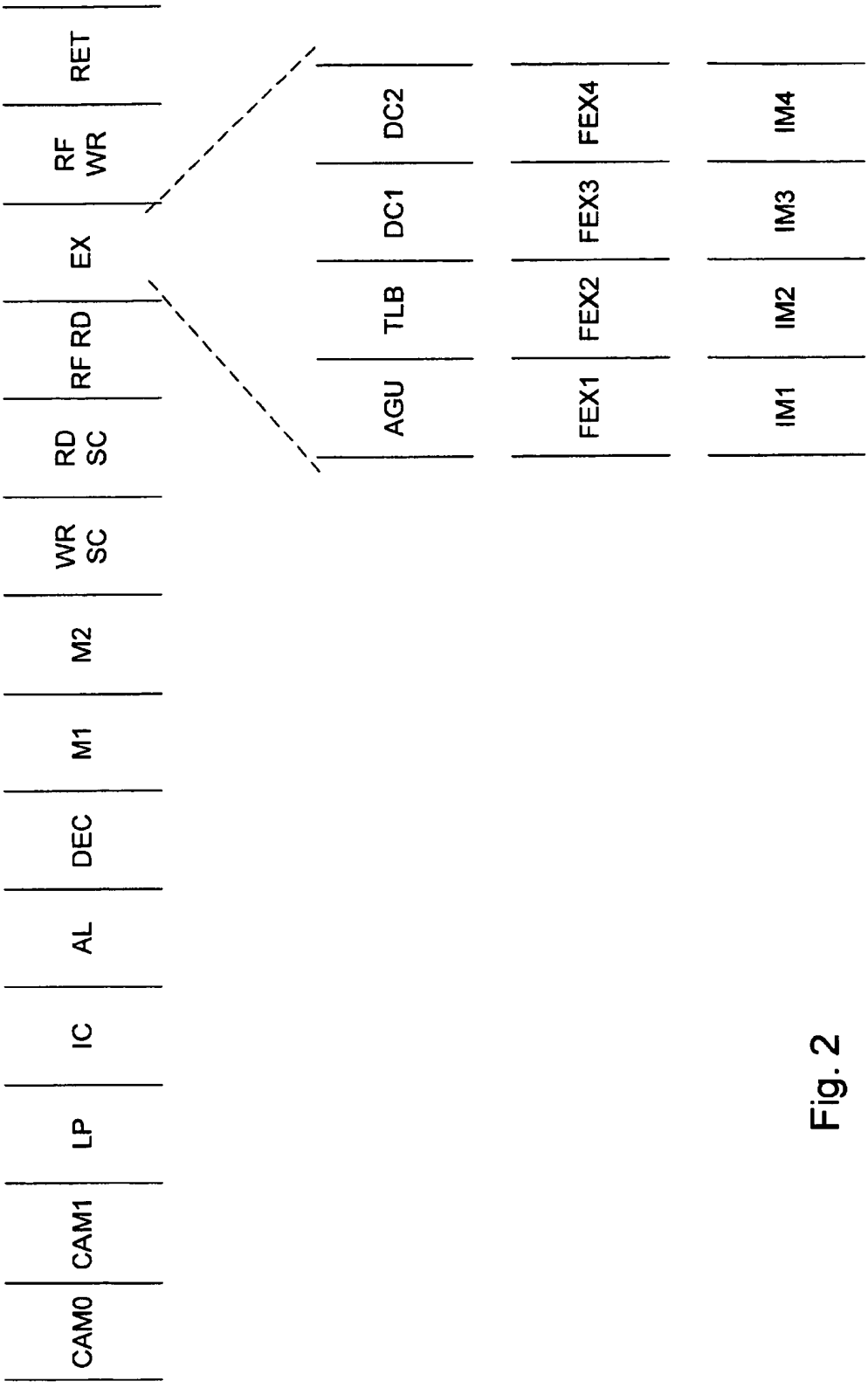

CAM0
CAM1
LP
IC
AL
DEC
M1
M2
WR SC
RD SC
RF RD
EX
RF WR
RET
AGU
TLB
DC1
DC2
FEX1
FEX2
FEX3
FEX4
IM1
IM2
IM3
IM4

Fig. 2

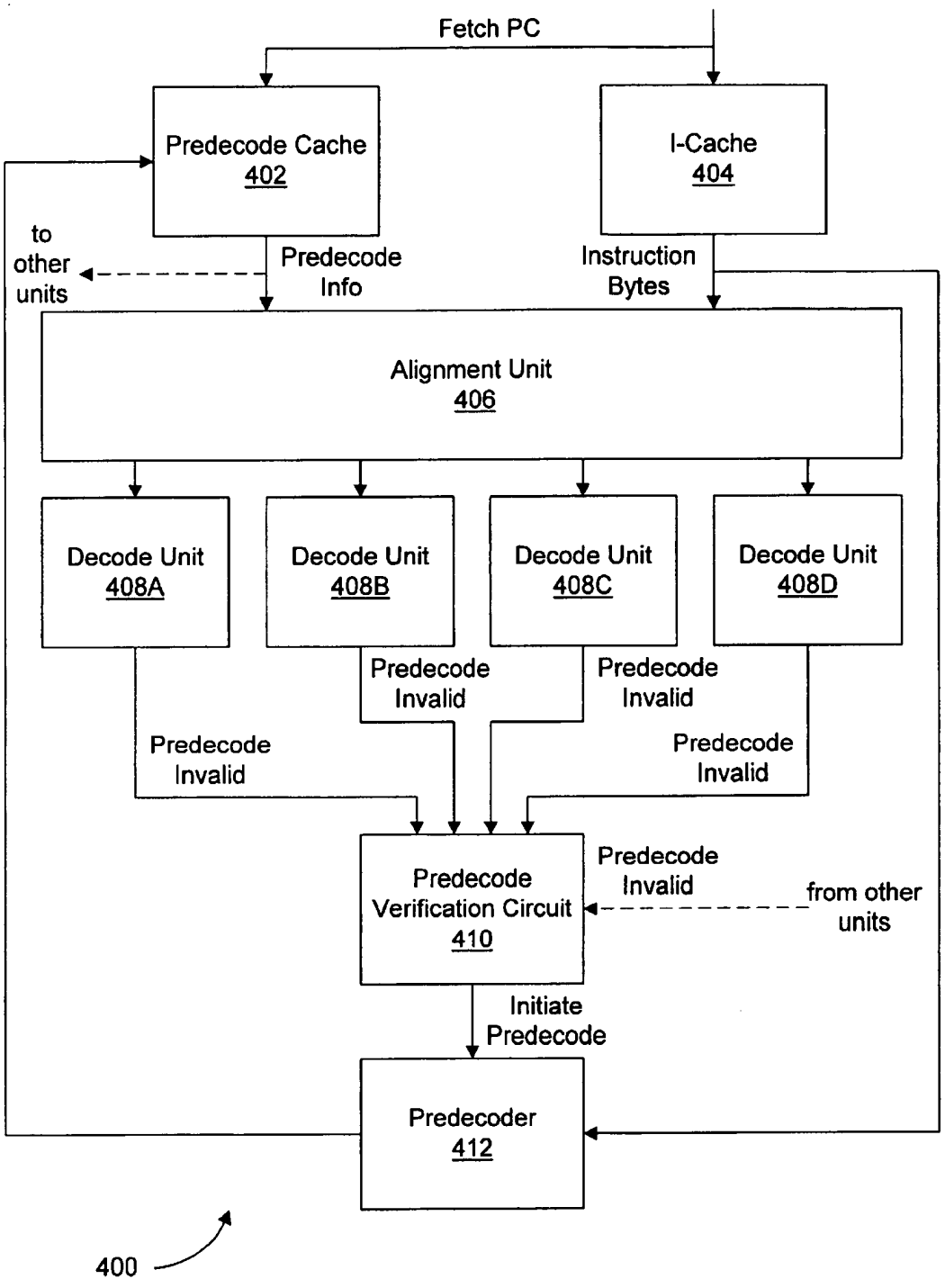

Fetch PC
Predecode Cache
402
I-Cache
404
to other units
Predecode Info
Instruction Bytes
Alignment Unit
406
Decode Unit
408A
Decode Unit
408B
Decode Unit
408C
Decode Unit
408D
Predecode Invalid
Predecode Invalid
Predecode Invalid
Predecode Invalid
Predecode Verification Circuit
410
Predecode Invalid
from other units
Initiate Predecode
Predecoder
412
400

Fig. 3

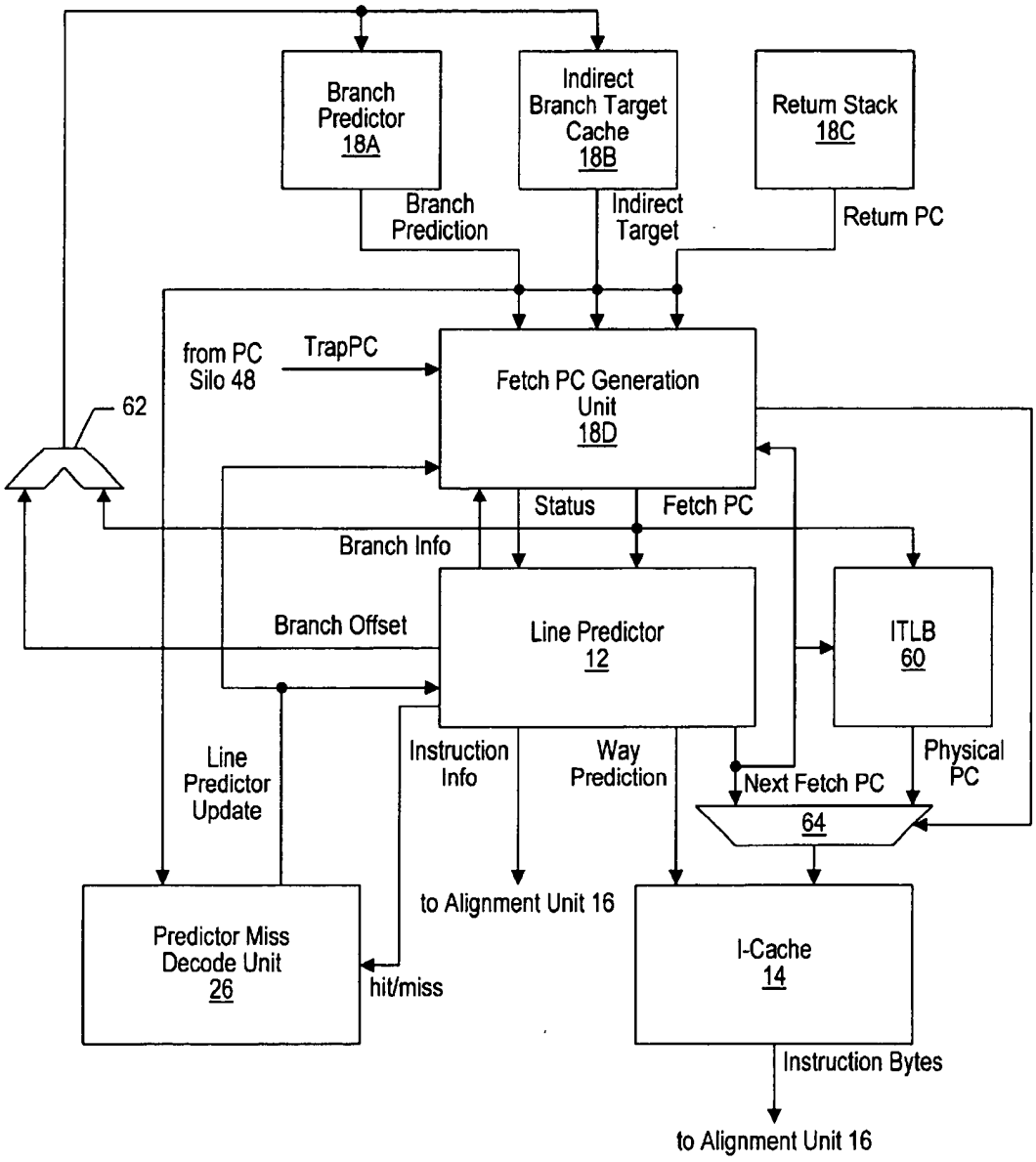

Branch Predictor 18A
Indirect Branch Target Cache 18B
Return Stack 18C
Branch Prediction
Indirect Target
Return PC
from PC Silo 48
TrapPC
Fetch PC Generation Unit 18D
62
Status
Fetch PC
Branch Info
Branch Offset
Line Predictor 12
ITLB 60
Line Predictor Update
Instruction Info
Way Prediction
Next Fetch PC
Physical PC
64
to Alignment Unit 16
Predictor Miss Decode Unit 26
hit/miss
I-Cache 14
Instruction Bytes
to Alignment Unit 16

Fig. 4

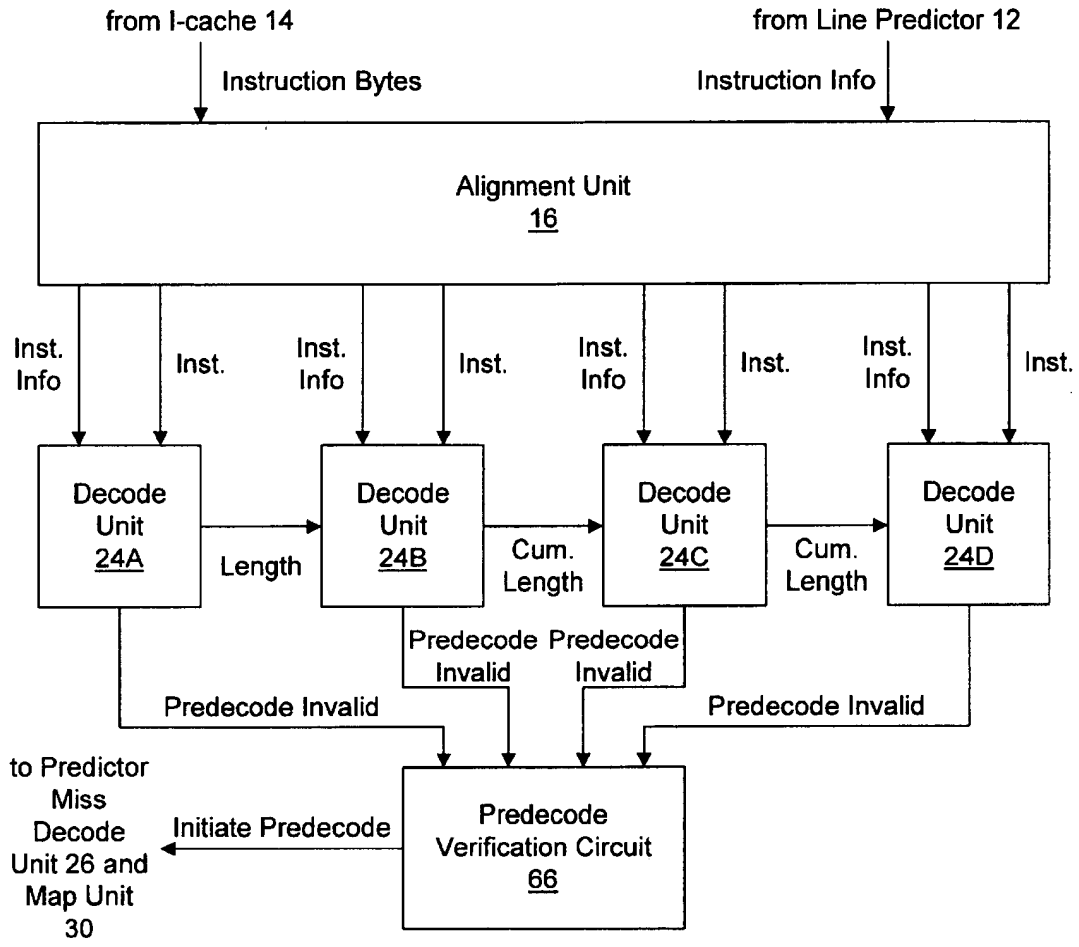

from I-cache 14
Instruction Bytes
from Line Predictor 12
Instruction Info
Alignment Unit
16
Inst. Info
Inst.
Inst. Info
Inst.
Inst. Info
Inst.
Inst. Info
Inst.
Decode Unit 24A
Length
Decode Unit 24B
Cum. Length
Decode Unit 24C
Cum. Length
Decode Unit 24D
Predecode Invalid
Predecode Invalid
Predecode Invalid
Predecode Invalid
to Predictor Miss Decode Unit 26 and Map Unit 30
Initiate Predecode
Predecode Verification Circuit
66

Fig. 5

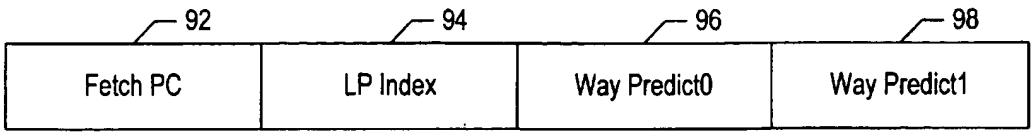


Fig. 6

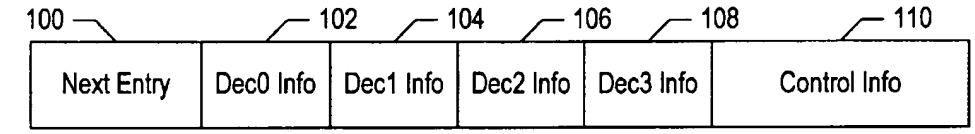


Fig. 7

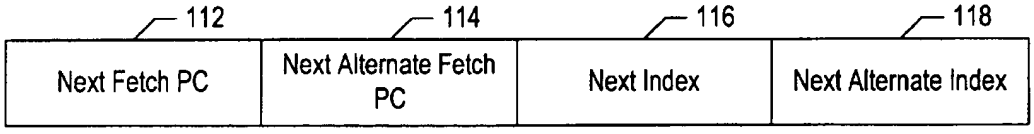


Fig. 8

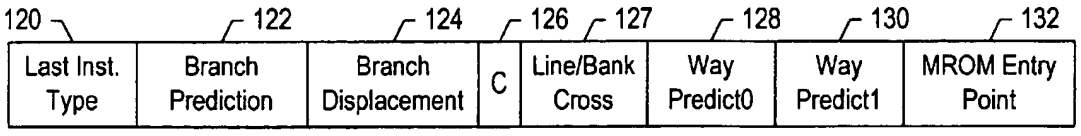


Fig. 9

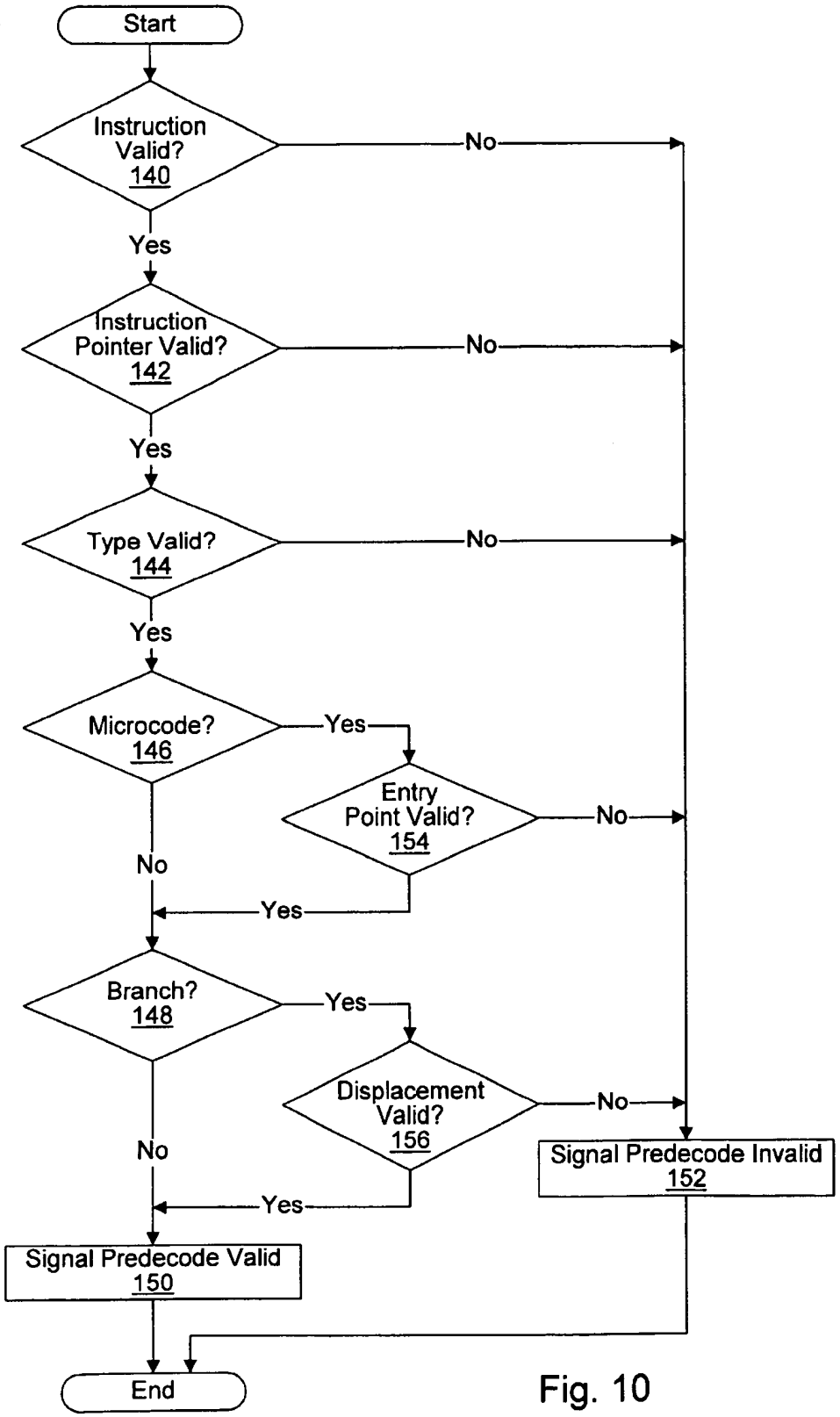

Start
Instruction Valid? 140
No
Yes
Instruction Pointer Valid? 142
No
Yes
Type Valid? 144
No
Yes
Microcode? 146
Yes
Entry Point Valid? 154
No
No
Yes
Branch? 148
Yes
Displacement Valid? 156
No
No
Yes
Signal Predecode Invalid 152
Signal Predecode Valid 150
End

Fig. 10

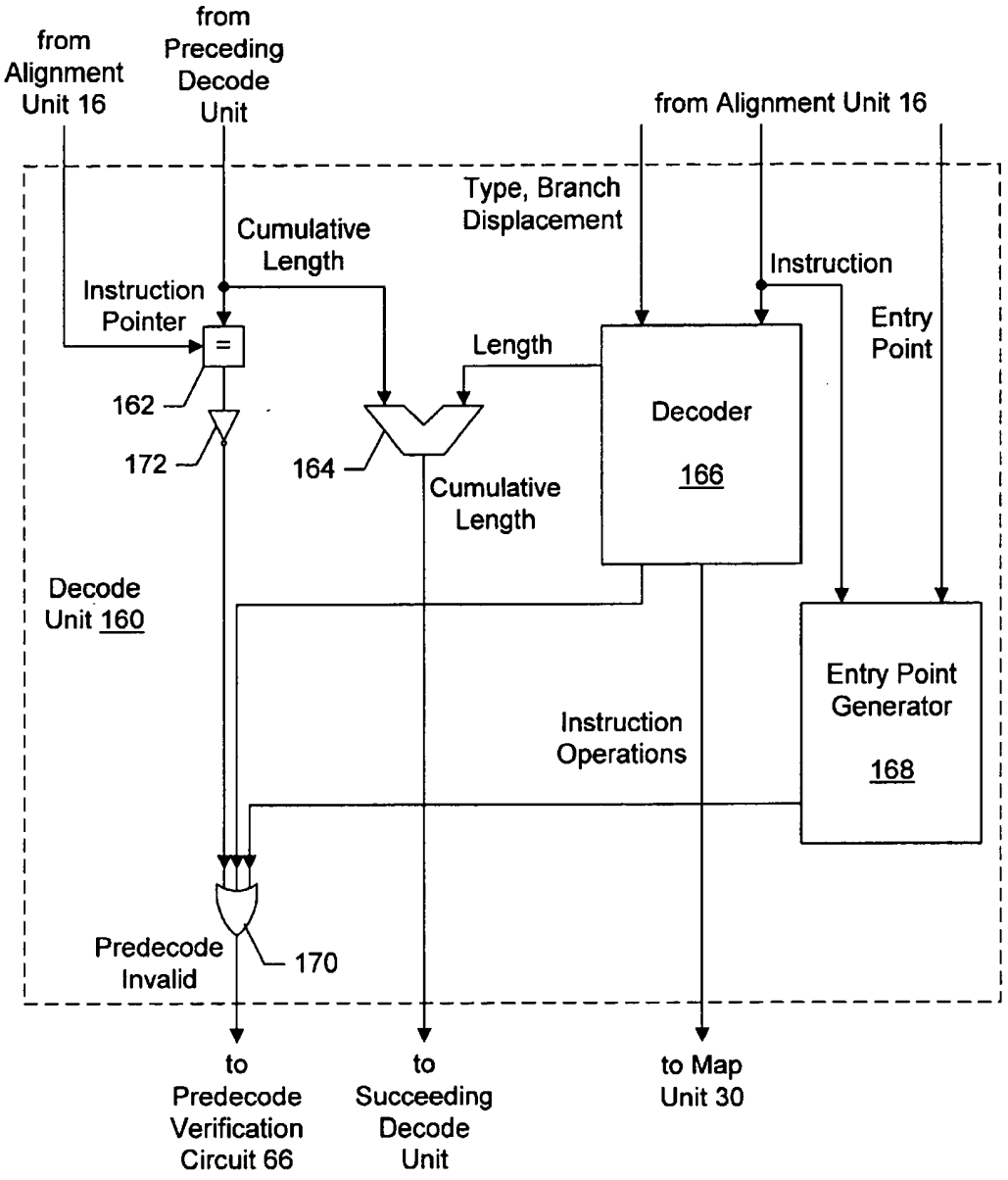

from Alignment Unit 16
from Preceding Decode Unit
from Alignment Unit 16
Type, Branch Displacement
Cumulative Length
Instruction
Instruction Pointer
Entry Point
=
Length
162
Decoder
166
172
164
Cumulative Length
Decode Unit 160
Entry Point Generator
168
Instruction Operations
Predecode Invalid
170
to Predecode Verification Circuit 66
to Succeeding Decode Unit
to Map Unit 30

Fig. 11

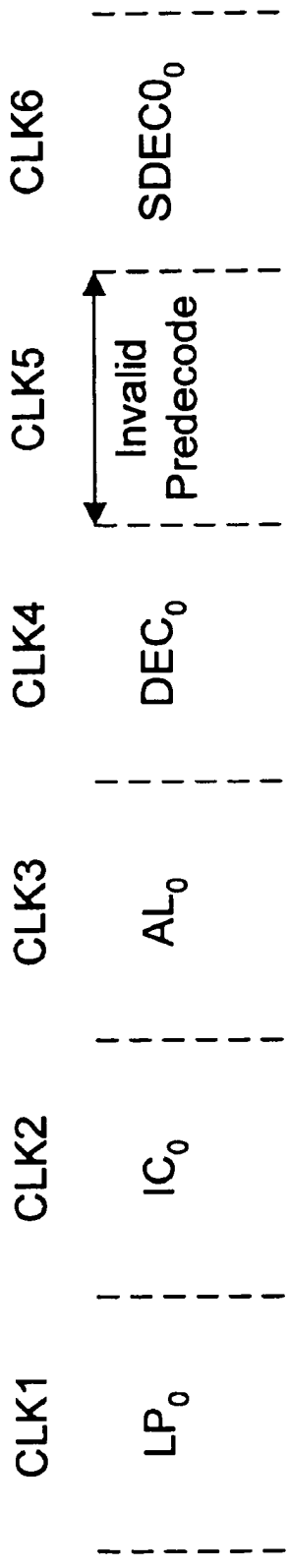

CLK6
CLK5
CLK4
CLK3
CLK2
CLK1
SDEC0$_0$
Invalid Predecode
DEC$_0$
AL$_0$
IC$_0$
LP$_0$

Fig. 12

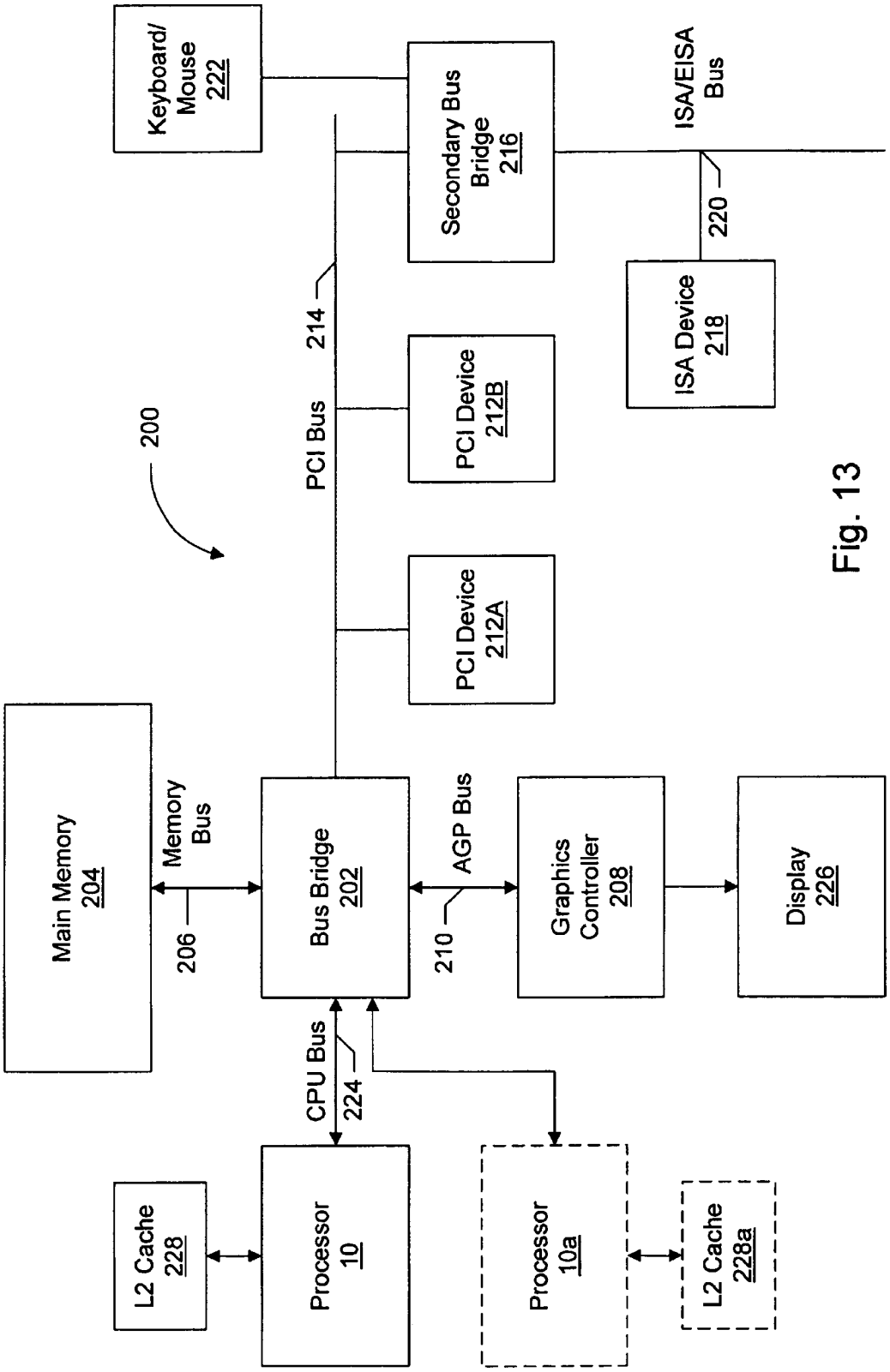

Keyboard/ Mouse 222
Secondary Bus Bridge 216
ISA/EISA Bus
220
ISA Device 218
PCI Bus 214
PCI Device 212B
PCI Device 212A
200
Main Memory 204
Memory Bus
206
Bus Bridge 202
AGP Bus
210
Graphics Controller 208
Display 226
CPU Bus
224
L2 Cache 228
Processor 10
Processor 10a
L2 Cache 228a

Fig. 13

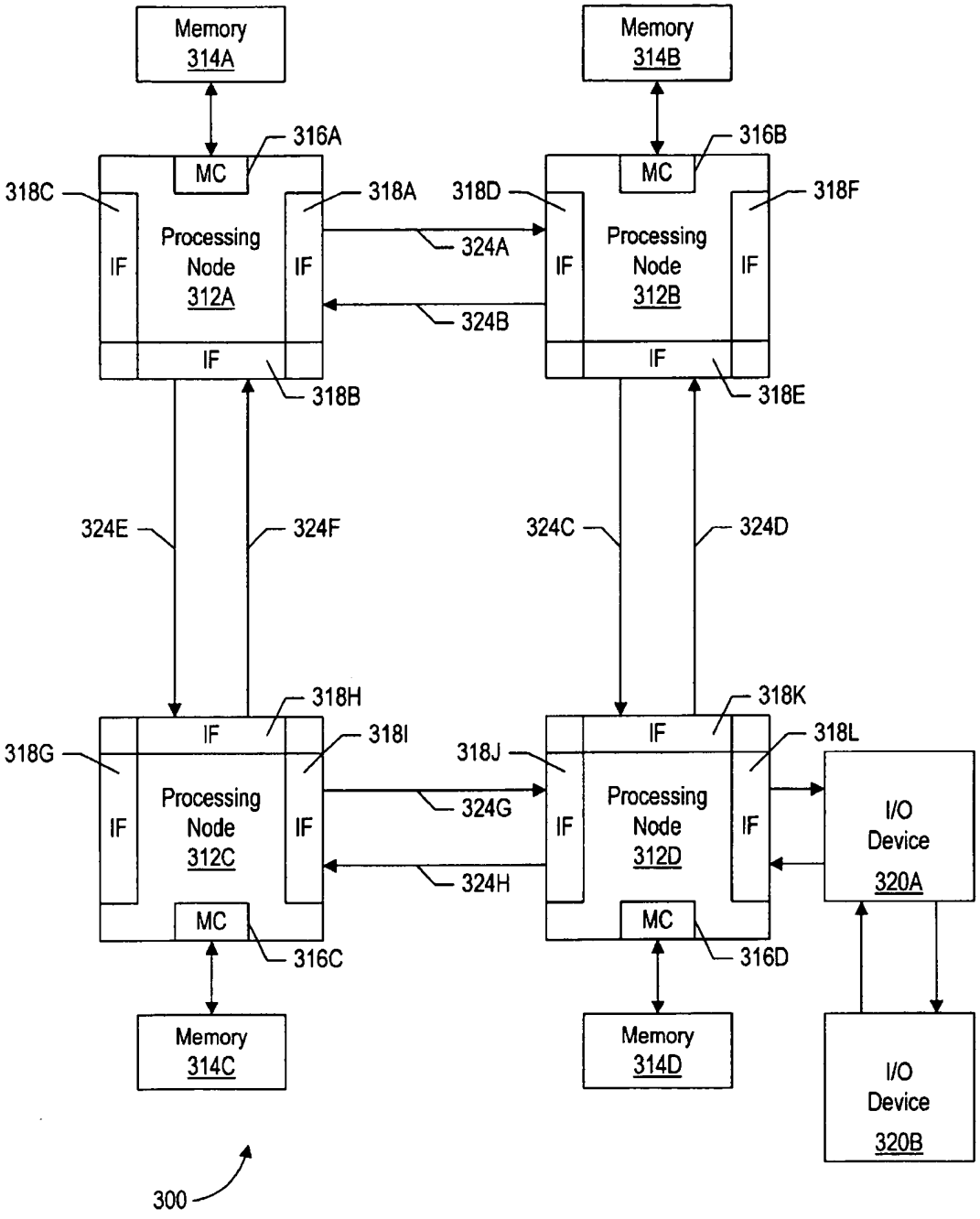

Memory
314A
Memory
314B
316A
316B
MC
MC
318C
318A
318D
318F
IF
Processing
Node
312A
IF
IF
Processing
Node
312B
IF
324A
324B
IF
IF
318B
318E
324E
324F
324C
324D
318H
318K
IF
IF
318G
318I
318J
318L
IF
Processing
Node
312C
IF
IF
Processing
Node
312D
IF
I/O
Device
320A
324G
324H
MC
MC
316C
316D
Memory
314C
Memory
314D
I/O
Device
320B
300

Fig. 14

PIPELINE ELEMENTS WHICH VERIFY PREDECODE INFORMATION

BACKGROUND OF THE INVENTION

1. Field of the Invention

This invention is related to the field of processors and, more particularly, to predecode mechanisms within processors.

2. Description of the Related Art

Superscalar processors achieve high performance by executing multiple instructions per clock cycle and by choosing the shortest possible clock cycle consistent with the design. As used herein, the term "clock cycle" refers to an interval of time accorded to various stages of an instruction processing pipeline within the processor. Storage devices (e.g. registers and arrays) capture their values according to the clock cycle. For example, a storage device may capture a value according to a rising or falling edge of a clock signal defining the clock cycle. The storage device then stores the value until the subsequent rising or falling edge of the clock signal, respectively. The term "instruction processing pipeline" is used herein to refer to the logic circuits employed to process instructions in a pipelined fashion. Although the pipeline may be divided into any number of stages at which portions of instruction processing are performed, instruction processing generally comprises fetching the instruction, decoding the instruction, executing the instruction, and storing the execution results in the destination identified by the instruction.

A popular instruction set architecture is the x86 instruction set architecture. Due to the widespread acceptance of the x86 instruction set architecture in the computer industry, superscalar processors designed in accordance with this architecture are becoming increasingly common. The x86 instruction set architecture specifies a variable byte-length instruction set in which different instructions may occupy differing numbers of bytes. For example, the 80386 and 80486 processors allow a particular instruction to occupy a number of bytes between 1 and 15. The number of bytes occupied depends upon the particular instruction as well as various addressing mode options for the instruction.

Because instructions are variable-length, locating instruction boundaries is complicated. The length of a first instruction must be determined prior to locating a second instruction subsequent to the first instruction within an instruction stream. However, the ability to locate multiple instructions within an instruction stream during a particular clock cycle is crucial to superscalar processor operation. As operating frequencies increase (i.e. as clock cycles shorten), it becomes increasingly difficult to locate multiple instructions simultaneously.

Various predecode schemes have been proposed in which a predecoder generates predecode information corresponding to a set of instruction bytes. The predecode information is stored and is fetched when the corresponding set of instruction bytes is fetched. Generally, the predecode information may be used to locate instructions within the set of instruction bytes and/or to quickly identify other attributes of the instructions being fetched. These other attributes may be used to direct further fetching or to direct additional hardware for accelerating the processing of the fetched instructions. Thus, predecoding may be effective for both fixed length and variable length instruction sets.

Typically, the predecode information is kept coherent with the instruction cache storing the instruction bytes, since the processor typically relies on the predecode information to rapidly and correctly process instructions. The predecode information may be stored in the instruction cache with the instruction bytes (and thus is deleted from the cache when the corresponding instruction bytes are deleted), or may be stored in a separate structure which has storage locations in a one-to-one correspondence with cache storage locations. By maintaining coherency with the instruction cache, the predecode information is never erroneously associated with a different set of instruction bytes.

It is desirable to allow for predecode information storage which is not coherent with the instruction cache. For example, it may be desirable to have fewer storage locations for predecode information than the instruction cache has storage locations for cache lines. Alternatively, it may be desirable to organize the predecode information in a different fashion than cache-line based storage. Accordingly, a processor which employs predecode information but does not actively maintain coherency between the predecode cache and the instruction cache is desired.

SUMMARY OF THE INVENTION

The problems outlined above are in large part solved by a processor as described herein. The processor includes an instruction cache and a predecode cache which is not actively maintained coherent with the instruction cache. The processor fetches instruction bytes from the instruction cache and predecode information from the predecode cache. Instructions are provided to a plurality of decode units based on the predecode information, and the decode units decode the instructions and verify that the predecode information corresponds to the instructions. More particularly, each decode unit may verify that a valid instruction was decoded, and that the instruction succeeds a preceding instruction decoded by another decode unit. Additionally, other units involved in the instruction processing pipeline stages prior to decode may verify portions of the predecode information. If the predecode information does not correspond to the fetched instructions, the predecode information may be corrected (either by predecoding the instruction bytes or by updating the predecode information, if the update may be determined without predecoding the instruction bytes). Advantageously, the predecode cache may be designed without attempting to match the instruction cache, and logic for maintaining coherency based on instruction cache updates may not be required.

In one particular embodiment, the predecode cache may be a line predictor which stores instruction pointers indexed by a portion of the fetch address. The line predictor may thus experience address aliasing, and predecode information may therefore not correspond to the instruction bytes. However, power may be conserved by not storing and comparing the entire fetch address.

Broadly speaking, a processor is contemplated. The processor comprises a predecode cache and one or more decode units coupled to receive predecode information from the instruction cache. The predecode cache is configured to store the predecode information, and is further configured to output the predecode information responsive to a fetch address. Each decode unit is further coupled to receive a portion of a plurality of instruction bytes fetched in response to the fetch address, and is configured to decode the portion. The decode units are configured to verify that the predecode information corresponds to the plurality of instruction bytes. Additionally, a computer system is contemplated including the processor and an input/output (I/O) device configured to

communicate between the computer system and another computer system to which the I/O device is couplable.

Furthermore, a method is contemplated. Predecode information is fetched from a predecode cache responsive to a fetch address. A plurality of instruction bytes are fetched responsive to the fetch address. The plurality of instruction bytes are decoded. The predecode information is verified as corresponding to the plurality of instruction bytes.

BRIEF DESCRIPTION OF THE DRAWINGS

Other objects and advantages of the invention will become apparent upon reading the following detailed description and upon reference to the accompanying drawings in which:

FIG. **1** is a block diagram of one embodiment of a processor.

FIG. **2** is a pipeline diagram illustrating pipeline stages of one embodiment of the processor shown in FIG. **1**.

FIG. **3** is a block diagram illustrating one embodiment of a predecode cache, I-cache, alignment unit, decode units, predecode verification circuit, and predecoder.

FIG. **4** is a block diagram of one embodiment of a branch prediction/fetch PC generation unit, line predictor, I-cache, and predictor miss decode unit shown in FIG. **1**.

FIG. **5** is a block diagram of one embodiment of an alignment unit and decode units shown in FIG. **1** and a predecode verification circuit shown in FIG. **1**.

FIG. **6** is a diagram illustrating one embodiment of an entry in a PC CAM within one embodiment of the line predictor.

FIG. **7** is a diagram illustrating one embodiment of an entry in an Index Table within one embodiment of the line predictor.

FIG. **8** is a diagram illustrating one embodiment of a next entry field shown in FIG. **7**.

FIG. **9** is a diagram illustrating one embodiment of a control information field shown in FIG. **7**.

FIG. **10** is a flowchart illustrating one embodiment of a decode unit verifying predecode information.

FIG. **11** is a block diagram of one embodiment of an exemplary decode unit.

FIG. **12** is a timing diagram illustrating initiation of predecode in response to verifying that the predecode information does not match the instructions.

FIG. **13** is a block diagram of a first exemplary computer system including the processor shown in FIG. **1**.

FIG. **14** is a block diagram of a second exemplary computer system including the processor shown in FIG. **1**.

While the invention is susceptible to various modifications and alternative forms, specific embodiments thereof are shown by way of example in the drawings and will herein be described in detail. It should be understood, however, that the drawings and detailed description thereto are not intended to limit the invention to the particular form disclosed, but on the contrary, the intention is to cover all modifications, equivalents and alternatives falling within the spirit and scope of the present invention as defined by the appended claims.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

Processor Overview

Turning now to FIG. **1**, a block diagram of one embodiment of a processor **10** is shown. Other embodiments are possible and contemplated. In the embodiment of FIG. **1**, processor **10** includes a line predictor **12**, an instruction cache (I-cache) **14**, an alignment unit **16**, a branch prediction/fetch PC generation unit **18**, a plurality of decode units **24A–24D**, a predictor miss decode unit **26**, a microcode unit **28**, a map unit **30**, a retire queue **32**, an architectural renames file **34**, a future file **20**, a scheduler **36**, an integer register file **38A**, a floating point register file **38B**, an integer execution core **40A**, a floating point execution core **40B**, a load/store unit **42**, a data cache (D-cache) **44**, an external interface unit **46**, and a PC silo **48**. Line predictor **12** is coupled to predictor miss decode unit **26**, branch prediction/fetch PC generation unit **18**, PC silo **48**, and alignment unit **16**. Line predictor **12** may also be coupled to I-cache **14**. I-cache **14** is coupled to alignment unit **16** and branch prediction/fetch PC generation unit **18**, which is further coupled to PC silo **48**. Alignment unit **16** is further coupled to predictor miss decode unit **26** and decode units **24A–24D**. Decode units **24A–24D** are further coupled to map unit **30**, and decode unit **24D** is coupled to microcode unit **28**. Map unit **30** is coupled to retire queue **32** (which is coupled to architectural renames file **34**), future file **20**, scheduler **36**, and PC silo **48**. Architectural renames file **34** is coupled to future file **20**. Scheduler **36** is coupled to register files **38A–38B**, which are further coupled to each other and respective execution cores **40A–40B**. Execution cores **40A–40B** are further coupled to load/store unit **42** and scheduler **36**. Execution core **40A** is further coupled to D-cache **44**. Load/store unit **42** is coupled to scheduler **36**, D-cache **44**, and external interface unit **46**. D-cache **44** is coupled to register files **38**. External interface unit **46** is coupled to an external interface **52** and to I-cache **14**. Elements referred to herein by a reference numeral followed by a letter will be collectively referred to by the reference numeral alone. For example, decode units **24A–24D** will be collectively referred to as decode units **24**.

In the embodiment of FIG. **1**, processor **10** employs a variable byte length, complex instruction set computing (CISC) instruction set architecture. For example, processor **10** may employ the x86 instruction set architecture (also referred to as IA-32). Other embodiments may employ other instruction set architectures including fixed length instruction set architectures and reduced instruction set computing (RISC) instruction set architectures. Certain features shown in FIG. **1** may be omitted in such architectures.

Branch prediction/fetch PC generation unit **18** is configured to provide a fetch address (fetch PC) to I-cache **14**, line predictor **12**, and PC silo **48**. Branch prediction/fetch PC generation unit **18** may include a suitable branch prediction mechanism used to aid in the generation of fetch addresses. In response to the fetch address, line predictor **12** provides alignment information corresponding to a plurality of instructions to alignment unit **16**, and may provide a next fetch address for fetching instructions subsequent to the instructions identified by the provided instruction information. The next fetch address may be provided to branch prediction/fetch PC generation unit **18** or may be directly provided to I-cache **14**, as desired. Branch prediction/fetch PC generation unit **18** may receive a trap address from PC silo **48** (if a trap is detected) and the trap address may comprise the fetch PC generated by branch prediction/fetch PC generation unit **18**. Otherwise, the fetch PC may be generated using the branch prediction information and information from line predictor **12**. Generally, line predictor **12** stores information corresponding to instructions previously speculatively fetched by processor **10**. In one embodiment, line predictor **12** includes 2K entries, each entry locating a

group of one or more instructions referred to herein as a "line" of instructions. The line of instructions may be concurrently processed by the instruction processing pipeline of processor **10** through being placed into scheduler **36**.

I-cache **14** is a high speed cache memory for storing instruction bytes. According to one embodiment I-cache **14** may comprise, for example, a 128 Kbyte, four way set associative organization employing 64 byte cache lines. However, any I-cache structure may be suitable (including direct-mapped structures).

Alignment unit **16** receives the instruction alignment information from line predictor **12** and instruction bytes corresponding to the fetch address from I-cache **14**. Alignment unit **16** selects instruction bytes into each of decode units **24A–24D** according to the provided instruction alignment information. More particularly, line predictor **12** provides an instruction pointer corresponding to each decode unit **24A–24D**. The instruction pointer locates an instruction within the fetched instruction bytes for conveyance to the corresponding decode unit **24A–24D**. In one embodiment, certain instructions may be conveyed to more than one decode unit **24A–24D**. Accordingly, in the embodiment shown, a line of instructions from line predictor **12** may include up to 4 instructions, although other embodiments may include more or fewer decode units **24** to provide for more or fewer instructions within a line.

Decode units **24A–24D** decode the instructions provided thereto, and each decode unit **24A–24D** generates information identifying one or more instruction operations (or ROPs) corresponding to the instructions. In one embodiment, each decode unit **24A–24D** may generate up to two instruction operations per instruction. As used herein, an instruction operation (or ROP) is an operation which an execution unit within execution cores **40A–40B** is configured to execute as a single entity. Simple instructions may correspond to a single instruction operation, while more complex instructions may correspond to multiple instruction operations. Certain of the more complex instructions may be implemented within microcode unit **28** as microcode routines (fetched from a read-only memory therein via decode unit **24D** in the present embodiment). Furthermore, other embodiments may employ a single instruction operation for each instruction (i.e. instruction and instruction operation may be synonymous in such embodiments).

PC silo **48** stores the fetch address and instruction information for each instruction fetch, and is responsible for redirecting instruction fetching upon exceptions (such as instruction traps defined by the instruction set architecture employed by processor **10**, branch mispredictions, and other microarchitecturally defined traps). PC silo **48** may include a circular buffer for storing fetch address and instruction information corresponding to multiple lines of instructions which may be outstanding within processor **10**. In response to retirement of a line of instructions, PC silo **48** may discard the corresponding entry. In response to an exception, PC silo **48** may provide a trap address to branch prediction/fetch PC generation unit **18**. Retirement and exception information may be provided by scheduler **36**. In one embodiment, PC silo **48** assigns a sequence number (R#) to each instruction to identify the order of instructions outstanding within processor **10**. Scheduler **36** may return R#s to PC silo **48** to identify instruction operations experiencing exceptions or retiring instruction operations.

Upon detecting a miss in line predictor **12**, alignment unit **16** routes the corresponding instruction bytes from I-cache **14** to predictor miss decode unit **26**. Predictor miss decode unit **26** decodes the instruction, enforcing any limits on a line of instructions as processor **10** is designed for (e.g. maximum number of instruction operations, maximum number of instructions, terminate on branch instructions, etc.). Upon terminating a line, predictor miss decode unit **26** provides the information to line predictor **12** for storage. It is noted that predictor miss decode unit **26** may be configured to dispatch instructions as they are decoded. Alternatively, predictor miss decode unit **26** may decode the line of instruction information and provide it to line predictor **12** for storage. Subsequently, the missing fetch address may be reattempted in line predictor **12** and a hit may be detected.

In addition to decoding instructions upon a miss in line predictor **12**, predictor miss decode unit **26** may be configured to decode instructions if the instruction information provided by line predictor **12** is invalid. In one embodiment, processor **10** does not attempt to keep information in line predictor **12** coherent with the instructions within I-cache **14** (e.g. when instructions are replaced or invalidate in I-cache **14**, the corresponding instruction information may not actively be invalidated). Decode units **24A–24D** may verify the instruction information provided, and may signal predictor miss decode unit **26** when invalid instruction information is detected. According to one particular embodiment, the following instruction operations are supported by processor **10**: integer (including arithmetic, logic, shift/rotate, and branch operations), floating point (including multimedia operations), and load/store.

The decoded instruction operations and source and destination register numbers are provided to map unit **30**. Map unit **30** is configured to perform register renaming by assigning physical register numbers (PR#s) to each destination register operand and source register operand of each instruction operation. The physical register numbers identify registers within register files **38A–38B**. Map unit **30** additionally provides an indication of the dependencies for each instruction operation by providing R#s of the instruction operations which update each physical register number assigned to a source operand of the instruction operation. Map unit **30** updates future file **20** with the physical register numbers assigned to each destination register (and the R# of the corresponding instruction operation) based on the corresponding logical register number. Additionally, map unit **30** stores the logical register numbers of the destination registers, assigned physical register numbers, and the previously assigned physical register numbers in retire queue **32**. As instructions are retired (indicated to map unit **30** by scheduler **36**), retire queue **32** updates architectural renames file **34** and frees any registers which are no longer in use. Accordingly, the physical register numbers in architectural register file **34** identify the physical registers storing the committed architectural state of processor **10**, while future file **20** represents the speculative state of processor **10**. In other words, architectural renames file **34** stores a physical register number corresponding to each logical register, representing the committed register state for each logical register. Future file **20** stores a physical register number corresponding to each logical register, representing the speculative register state for each logical register.

The line of instruction operations, source physical register numbers, and destination physical register numbers are stored into scheduler **36** according to the R#s assigned by PC silo **48**. Furthermore, dependencies for a particular instruction operation may be noted as dependencies on other instruction operations which are stored in the scheduler. In one embodiment, instruction operations remain in scheduler **36** until retired.

Scheduler **36** stores each instruction operation until the dependencies noted for that instruction operation have been satisfied. In response to scheduling a particular instruction operation for execution, scheduler **36** may determine at which clock cycle that particular instruction operation will update register files **38A–38B**. Different execution units within execution cores **40A–40B** may employ different numbers of pipeline stages (and hence different latencies). Furthermore, certain instructions may experience more latency within a pipeline than others. Accordingly, a countdown is generated which measures the latency for the particular instruction operation (in numbers of clock cycles). Scheduler **36** awaits the specified number of clock cycles (until the update will occur prior to or coincident with the dependent instruction operations reading the register file), and then indicates that instruction operations dependent upon that particular instruction operation may be scheduled. It is noted that scheduler **36** may schedule an instruction once its dependencies have been satisfied (i.e. out of order with respect to its order within the scheduler queue).

Integer and load/store instruction operations read source operands according to the source physical register numbers from register file **38A** and are conveyed to execution core **40A** for execution. Execution core **40A** executes the instruction operation and updates the physical register assigned to the destination within register file **38A**. Additionally, execution core **40A** reports the R# of the instruction operation and exception information regarding the instruction operation (if any) to scheduler **36**. Register file **38B** and execution core **40B** may operate in a similar fashion with respect to floating point instruction operations (and may provide store data for floating point stores to load/store unit **42**).

In one embodiment, execution core **40A** may include, for example, two integer units, a branch unit, and two address generation units (with corresponding translation lookaside buffers, or TLBs). Execution core **40B** may include a floating point/multimedia multiplier, a floating point/multimedia adder, and a store data unit for delivering store data to load/store unit **42**. Other configurations of execution units are possible.

Load/store unit **42** provides an interface to D-cache **44** for performing memory operations and for scheduling fill operations for memory operations which miss D-cache **44**. Load memory operations may be completed by execution core **40A** performing an address generation and forwarding data to register files **38A–38B** (from D-cache **44** or a store queue within load/store unit **42**). Store addresses may be presented to D-cache **44** upon generation thereof by execution core **40A** (directly via connections between execution core **40A** and D-Cache **44**). The store addresses are allocated a store queue entry. The store data may be provided concurrently, or may be provided subsequently, according to design choice. Upon retirement of the store instruction, the data is stored into D-cache **44** (although there may be some delay between retirement and update of D-cache **44**). Additionally, load/store unit **42** may include a load/store buffer for storing load/store addresses which miss D-cache **44** for subsequent cache fills (via external interface unit **46**) and re-attempting the missing load/store operations. Load/store unit **42** is further configured to handle load/store memory dependencies.

D-cache **44** is a high speed cache memory for storing data accessed by processor **10**. While D-cache **44** may comprise any suitable structure (including direct mapped and set-associative structures), one embodiment of D-cache **44** may comprise a 128 Kbyte, 2 way set associative cache having 64 byte lines.

External interface unit **46** is configured to communicate to other devices via external interface **52**. Any suitable external interface **52** may be used, including interfaces to L2 caches and an external bus or buses for connecting processor **10** to other devices. External interface unit **46** fetches fills for I-cache **16** and D-cache **44**, as well as writing discarded updated cache lines from D-cache **44** to the external interface. Furthermore, external interface unit **46** may perform non-cacheable reads and writes generated by processor **10** as well.

Turning next to FIG. **2**, an exemplary pipeline diagram illustrating an exemplary set of pipeline stages which may be employed by one embodiment of processor **10** is shown. Other embodiments may employ different pipelines, pipelines including more or fewer pipeline stages than the pipeline shown in FIG. **2**. The stages shown in FIG. **2** are delimited by vertical dashed lines. Each stage is one clock cycle of a clock signal used to clock storage elements (e.g. registers, latches, flops, and the like) within processor **10**.

As illustrated in FIG. **2**, the exemplary pipeline includes a CAM0 stage, a CAM1 stage, a line predictor (LP) stage, an instruction cache (IC) stage, an alignment (AL) stage, a decode (DEC) stage, a map1 (M1) stage, a map2 (M2) stage, a write scheduler (WR SC) stage, a read scheduler (RD SC) stage, a register file read (RF RD) stage, an execute (EX) stage, a register file write (RF WR) stage, and a retire (RET) stage. Some instructions utilize multiple clock cycles in the execute state. For example, memory operations, floating point operations, and integer multiply operations are illustrated in exploded form in FIG. **2**. Memory operations include an address generation (AGU) stage, a translation (TLB) stage, a data cache 1 (DC1) stage, and a data cache 2 (DC2) stage. Similarly, floating point operations include up to four floating point execute (FEX1–FEX4) stages, and integer multiplies include up to four (IM1–IM4) stages.

During the CAM0 and CAM1 stages, line predictor **12** compares the fetch address provided by branch prediction/fetch PC generation unit **18** to the addresses of lines stored therein. Additionally, the fetch address is translated from a virtual address (e.g. a linear address in the x86 architecture) to a physical address during the CAM0 and CAM1 stages. In response to detecting a hit during the CAM0 and CAM1 stages, the corresponding line information is read from the line predictor during the line predictor stage. Also, I-cache **14** initiates a read (using the physical address) during the line predictor stage. The read completes during the instruction cache stage.

It is noted that, while the pipeline illustrated in FIG. **2** employs two clock cycles to detect a hit in line predictor **12** for a fetch address, other embodiments may employ a single clock cycle (and stage) to perform this operation. Moreover, in one embodiment, line predictor **12** provides a next fetch address for I-cache **14** and a next entry in line predictor **12** for a hit, and therefore the CAM0 and CAM1 stages may be skipped for fetches resulting from a previous hit in line predictor **12**.

Instruction bytes provided by I-cache **14** are aligned to decode units **24A–24D** by alignment unit **16** during the alignment stage in response to the corresponding line information from line predictor **12**. Decode units **24A–24D** decode the provided instructions, identifying ROPs corresponding to the instructions as well as operand information during the decode stage. Map unit **30** generates ROPs from the provided information during the map1 stage, and performs register renaming (updating future file **20**). During the map2 stage, the ROPs and assigned renames are recorded in retire queue **32**. Furthermore, the ROPs upon which each

ROP is dependent are determined. Each ROP may be register dependent upon earlier ROPs as recorded in the future file, and may also exhibit other types of dependencies (e.g. dependencies on a previous serializing instruction, etc.)

The generated ROPs are written into scheduler **36** during the write scheduler stage. Up until this stage, the ROPs located by a particular line of information flow through the pipeline as a unit. However, subsequent to be written into scheduler **36**, the ROPs may flow independently through the remaining stages, at different times Generally, a particular ROP remains at this stage until selected for execution by scheduler **36** (e.g. after the ROPs upon which the particular ROP is dependent have been selected for execution, as described above). Accordingly, a particular ROP may experience one or more clock cycles of delay between the write scheduler write stage and the read scheduler stage. During the read scheduler stage, the particular ROP participates in the selection logic within scheduler **36**, is selected for execution, and is read from scheduler **36**. The particular ROP then proceeds to read register file operands from one of register files **38A–38B** (depending upon the type of ROP) in the register file read stage.

The particular ROP and operands are provided to the corresponding execution core **40A** or **40B**, and the instruction operation is performed on the operands during the execution stage. As mentioned above, some ROPs have several pipeline stages of execution. For example, memory instruction operations (e.g. loads and stores) are executed through an address generation stage (in which the data address of the memory location accessed by the memory instruction operation is generated), a translation stage (in which the virtual data address provided by the address generation stage is translated) and a pair of data cache stages in which D-cache **44** is accessed. Floating point operations may employ up to 4 clock cycles of execution, and integer multiplies may similarly employ up to 4 clock cycles of execution.

Upon completing the execution stage or stages, the particular ROP updates its assigned physical register during the register file write stage. Finally, the particular ROP is retired after each previous ROP is retired (in the retire stage). Again, one or more clock cycles may elapse for a particular ROP between the register file write stage and the retire stage. Furthermore, a particular ROP may be stalled at any stage due to pipeline stall conditions, as is well known in the art.

Predecode Verification

Turning now to FIG. **3**, a block diagram of a generalized apparatus **400** for storing predecode information without maintaining coherency with the instruction cache and for verifying the correctness of the predecode information during instruction processing within the pipeline is shown. Other embodiments are possible and contemplated. The generalized apparatus shown in FIG. **3** may be used in any type of processor, including processor **10** shown in FIG. **1**. A more specific implementation employed by one embodiment of processor **10** is described in more detail below. In the embodiment of FIG. **3**, apparatus **400** includes a predecode cache **402**, an I-cache **404**, an alignment unit **406**, a plurality of decode units **408A–408D**, a predecode verification circuit **410**, and a predecoder **412**. Predecode cache **402** and I-cache **404** are coupled to receive a fetch address (or fetch PC), and are coupled to alignment unit **406**. Alignment unit **406** is coupled to decode units **408A–408D**, which are further coupled to predecode verification circuit **410**. Predecode verification circuit **410** is optionally coupled to other units involved in the processing of instructions, and is coupled to predecoder **412**. Predecoder **412** is coupled to receive instruction bytes from I-cache **404** and to provide predecode information to predecode cache **402**.

Generally, I-cache **404** provides instruction bytes in response to the fetch address, and predecode cache **402** provides predecode information. The predecode information may or may not correspond to the instruction bytes (also referred to herein as the predecode information being invalid, even though the predecode information may be valid for a different set of instruction bytes, or the predecode information being incorrect). For example, the predecode cache **402** may include fewer storage locations than the I-cache **404**, and hence more than one cache line may map to the same storage location within predecode cache **402**. To further reduce storage, the predecode cache **402** may be a tagless table (i.e. no address may be stored to compare to the fetch address to determine if the corresponding predecode information corresponds to the fetched instruction bytes). On the other hand, predecode cache **402** may perform a partial compare of the fetch address to a partial tag stored in the predecode cache **402**. Accordingly, predecode data corresponding to an address alias of the fetch address may be fetched.

Alignment unit **406** receives the predecode information and the instruction bytes, and aligns instruction bytes to decode units **408A–408D** in response to the predecode information. More particularly, alignment unit **406** presumes that the predecode information corresponds to the instruction bytes and aligns portions of the instruction bytes identified as instructions by the predecode information to the decode units **408A–408D**. Additionally, alignment unit **406** may provide the predecode information to decode units **408A–408D**. The predecode information used to align a particular instruction to a particular decode unit **408A–408D** may be provided to that particular decode unit. Additionally, predecode information which identifies additional attributes of an instruction (beyond the location of the instructions within the instruction bytes) may be provided to the decode unit **408A–408D** which receives that instruction. Alternatively, the additional attribute information may be provided to each decode unit **408A–408D**, and the decode unit receiving the corresponding instruction may verify the additional attribute information.

Decode units **408A–408D** decode the received instruction bytes and provide decoded instruction operations to the subsequent pipeline stages (not shown). Additionally, since it is possible that the predecode information is incorrect for the fetched instruction bytes, decode units **408A–408D** verify the predecode information against the instruction bytes received. Predecode information is incorrect if the instruction bytes received by one of the decode units **408A–408D** is not a valid instruction. Furthermore, the predecode information may be incorrect if the additional instruction attributes defined by the predecode information (if any) do not correspond to the decoded instructions. Decode units **408A–408D** may verify many of the additional attributes as well.

Decode units **408A–408D** report the results of verifying the predecode information to predecode verification circuit **410** using the predecode invalid signals illustrated in FIG. **3**. Each decode unit **408A–408D** asserts the corresponding predecode invalid signal if that decode unit **408A–408D** detects one or more inconsistencies between the predecode information and the instruction bytes provided by alignment unit **406** to that decode unit **408A–408D**. If a particular decode unit **408A–408D** does not detect that the predecode data is incorrect with respect to the instruction bytes received by that particular decode unit **408A–408D**, the

particular decode unit **408A–408D** deasserts the corresponding predecode invalid signal. Accordingly, if one or more of the predecode invalid signals are asserted, predecode verification circuit **410** signals predecoder **412** to predecode the corresponding instruction bytes. Predecoder **412** decodes the instruction bytes and generates predecode information corresponding to the instruction bytes. The generated predecode information is then updated into predecode cache **402**, thereby becoming available for the subsequent fetch of the instruction bytes. Predecoder **412** may also be configured to dispatch the instructions within the instruction bytes to subsequent pipeline stages, if desired.

Optionally, one or more other units involved in the processing of instructions may be coupled to receive a portion or all of the predecode information, and may be configured to verify the received predecode information as well. These other units report the results of verifying the received predecode information using predecode invalid signals as well. Alternatively, the other units may correct the predecode information in predecode cache **402** directly, if the correction may be determined without predecoding the instruction bytes.

It is noted that apparatus **400** and the processor in which it is implemented may be pipelined. Accordingly, predecode verification circuit **410** may be configured to indicate to predecoder **412** which of the sets of instruction bytes which are in-flight within the pipeline is to be predecoded, if predecoding may be initiated while the corresponding set of instruction bytes is at different pipeline stages (depending upon which unit detects the incorrect predecode information). Furthermore, if the verification of predecode information is delayed with respect to the completion of decoding of instructions, other pipeline stages may be informed of the determination that the predecode information is invalid, so that those instructions may be invalidated.

As used herein, the term "predecode information" refers to information generated via predecoding of instruction bytes. Predecode information may include information locating instructions within the instruction bytes. For example, the embodiment described below uses instruction pointers to identify instructions within the instruction bytes. Other embodiments may use other methods to locate instructions, such as a start bit and an end bit for each instruction byte. If the start bit is set, the instruction byte is the start of an instruction. If the end bit is set, the instruction byte is the end of an instruction. Yet another embodiment stores an instruction length for each byte, which indicates the length of the instruction if the corresponding byte is the start of an instruction. Any suitable encoding for locating instructions may be used. Predecode information may include additional attributes. An example of additional attributes is described further below. As another example, one embodiment may include the number of ROPs per instruction. Yet another embodiment may include information indicating whether or not each instruction is a microcode instruction. Any suitable additional attributes may be included according to design choice.

As used herein, the term "invalid instruction" refers to a group of instruction bytes which do not form a valid instruction. In other words, the combination of bytes is not defined as an instruction according to the instruction set architecture employed by the processor. As used herein, the term "assert" refers to providing a logically true value for a signal or a bit. A signal or bit may be asserted if it conveys a value indicative of a particular condition. Conversely, a signal or bit may be "deasserted" if it conveys a value indicative of a lack of a particular condition. A signal or bit may be defined to be asserted when it conveys a logical zero value or, conversely, when it conveys a logical one value, and the signal or bit may be defined as deasserted when the opposite logical value is conveyed.

It is noted that, while predecode invalid signals are described as being asserted when predecode information does not correspond to the fetched instruction bytes, predecode valid signals could instead be used. The predecode valid signals would be asserted when the predecode information does correspond to the fetched instruction bytes.

Turning now to FIG. **4**, a block diagram illustrating one embodiment of branch prediction/fetch PC generation unit **18**, line predictor **12**, I-cache **14**, predictor miss decode unit **26**, an instruction TLB (ITLB) **60**, an adder **62**, and a fetch address mux **64**. Other embodiments are possible and contemplated. In the embodiment of FIG. **4**, branch prediction/fetch PC generation unit **18** includes a branch predictor **18A**, an indirect branch target cache **18B**, a return stack **18C**, and fetch PC generation unit **18D**. Branch predictor **18A** and indirect branch target cache **18B** are coupled to receive the output of adder **62**, and are coupled to fetch PC generation unit **18D** and predictor miss decode unit **26**. Return stack **18C** is coupled to fetch PC generation unit **18D** and predictor miss decode unit **26**. Fetch PC generation unit **18D** is coupled to receive a trap PC from PC silo **48**, and is further coupled to ITLB **60**, line predictor **12**, adder **62**, and fetch address mux **64**. ITLB **60** is further coupled to fetch address mux **64**, which is coupled to I-cache **14**. Line predictor **12** is coupled to I-cache **14**, predictor miss decode unit **26**, ITLB **60**, adder **62**, and fetch address mux **64**.

The embodiment shown in FIGS. **4** and **5** may be a particular implementation of hardware for verifying predecode information. Line predictor **12** may be an example of a predecode cache similar to predecode cache **402**, with the instruction information stored in a line predictor entry being an example of predecode information. I-cache **14** may be an example of an I-cache similar to I-cache **404**. Alignment unit **16** may be an example of an alignment unit similar to alignment unit **406**. Decode units **24A–24D** may be an example of decode units similar to decode units **408A–408D**. Predecode verification circuit **66** may be an example of a predecode verification circuit similar to predecode verification circuit **410**. Predictor miss decode unit **26** may be an example of a predecoder similar to predecoder **412**. Additionally, branch predictor **18A**, indirect branch target cache **18B**, return stack **18C**, ITLB **60**, and I-cache **14** may be examples of other units which update predecode information (instruction information within line predictor **12**) directly.

Generally, fetch PC generation unit **18D** generates a fetch address (fetch PC) for instructions to be fetched. The fetch address is provided to line predictor **12**, TLB **60**, and adder **62** (as well as PC silo **48**, as shown in FIG. **1**). Line predictor **12** compares the fetch address to fetch addresses stored therein to determine if a line predictor entry corresponding to the fetch address exists within line predictor **12**. If a corresponding line predictor entry is found, the instruction pointers stored in the line predictor entry are provided to alignment unit **16** (as well as other instruction information within the line predictor entry for verification by decode units **24A–24D**). In parallel with line predictor **12** searching the line predictor entries, ITLB **60** translates the fetch address (which is a virtual address in the present embodiment) to a physical address (physical PC) for access to I-cache **14**. ITLB **60** provides the physical address to fetch address mux **64**, and fetch PC generation unit **18D** controls mux **64** to select the physical address. I-cache **14** reads

instruction bytes corresponding to the physical address and provides the instruction bytes to alignment unit **16**.

In the present embodiment, each line predictor entry also provides a next fetch address (next fetch PC). The next fetch address is provided to mux **64**, and fetch PC generation unit **18D** selects the address through mux **64** to access I-cache **14** in response to line predictor **12** detecting a hit. In this manner, the next fetch address may be more rapidly provided to I-cache **14** as long as the fetch addresses continue to hit in the line predictor. The line predictor entry may also include an indication of the next line predictor entry within line predictor **12** (corresponding to the next fetch address) to allow line predictor **12** to fetch the line predictor entry corresponding to the next fetch address. Accordingly, as long as fetch addresses continue to hit in line predictor **12**, fetching of lines of instructions may be initiated from the line predictor stage of the pipeline shown in FIG. **2**. Traps initiated by PC silo **48** (in response to scheduler **36**), a disagreement between the prediction made by line predictor **12** for the next fetch address and the next fetch address generated by fetch PC generation unit **18D** (described below) and page crossings (described below) may cause line predictor **12** to search for the fetch address provided by fetch PC generation unit **18D**, and may also cause fetch PC generation unit **18D** to select the corresponding physical address provided by ITLB **60**.

Even while next fetch addresses are being generated by line predictor **12** and are hitting in line predictor **12**, fetch PC generation unit **18D** continues to generate fetch addresses for logging by PC silo **48**. Additionally, the next fetch addresses may be translated by ITLB **60**, which may then verify that the physical next fetch address matches the next fetch address provided by line predictor **12**. More particularly, ITLB **60** may compare the next fetch address from line predictor **12** to the corresponding translated next fetch address. If a mismatch is detected, ITLB **60** may provide the corrected next fetch address to line predictor **12** to update the line predictor entry including the next fetch address. Additionally, instruction bytes are fetched from I-cache **14** using the corrected next fetch address.

Furthermore, fetch PC generation unit **18D** may verify the next fetch addresses provided by line predictor **12** via the branch predictors **18A–18C**. The line predictor entries within line predictor **12** identify the terminating instruction within the line of instructions by type, and line predictor **12** transmits the type information to fetch PC generation unit **18D** as well as the predicted direction of the terminating instruction (branch info in FIG. **4**). Furthermore, for branches forming a target address via a branch displacement included within the branch instruction, line predictor **12** may provide an indication of the branch displacement. For purposes of verifying the predicted next fetch address, the terminating instruction may be a conditional branch instruction, an indirect branch instruction, or a return instruction.

If the terminating instruction is a conditional branch instruction or an indirect branch instruction, line predictor **12** generates a branch offset from the current fetch address to the branch instruction by examining the instruction pointers in the line predictor entry. The branch offset is added to the current fetch address by adder **62**, and the address is provided to branch predictor **18A** and indirect branch target cache **18B**. Branch predictor **18A** is used for conditional branches, and indirect branch target cache **18B** is used for indirect branches.

Generally, branch predictor **18A** is a mechanism for predicting conditional branches based on the past behavior of conditional branches. More particularly, the address of the branch instruction is used to index into a table of branch predictions (e.g., two bit saturating counters which are incremented for taken branches and decremented for not-taken branches, and the most significant bit is used as a taken/not-taken prediction). The table is updated based on past executions of conditional branch instructions, as those branch instructions are retired or become non-speculative. In one particular embodiment, two tables are used (each having 16K entries of two bit saturating counters). The tables are indexed by an exclusive OR of recent branch prediction history and the least significant bits of the branch address, and each table provides a prediction. A third table (comprising 4K entries of two bit saturating selector counters) stores a selector between the two tables, and is indexed by the branch address directly. The selector picks one of the predictions provided by the two tables as the prediction for the conditional branch instruction. The third table may be updated according to each execution of a branch instruction. The entry in the third table indexed by the branch instruction may store a two bit saturating counter which may be incremented if the branch is taken and decremented if the branch is not taken. Other embodiments may employ different configurations and different numbers of entries. Using the three table structure, aliasing of branches having the same branch history and least significant address bits (but different most significant address bits) may be alleviated.

In response to the address provided by adder **62**, branch predictor **18A** provides a branch prediction. Fetch PC generation unit **18D** compares the prediction to the prediction recorded in the line predictor entry. If the predictions do not match, fetch PC generation unit **18D** signals (via status lines shown in FIG. **4**) line predictor **12**. Additionally, fetch PC generation unit **18D** generates a fetch address based on the prediction from branch predictor **18A** (either the branch target address generated in response to the branch displacement, or the sequential address). More particularly, the branch target address in the x86 instruction set architecture may be generated by adding the sequential address and the branch displacement. Other instruction set architectures may add the address of the branch instruction to the branch displacement.

In one embodiment, line predictor **12** stores a next alternate fetch address (and alternate indication of the next line predictor entry) in each line predictor entry. If fetch PC generation unit **18D** signals a mismatch between the prediction recorded in a particular line predictor entry and the prediction from branch predictor **18A**, line predictor **12** may swap the next fetch address and next alternate fetch address. In this manner, the line predictor entry may be updated to reflect the actual execution of branch instructions (recorded in branch predictor **18A**). The line predictor is thereby trained to match recent branch behavior, without requiring that the line predictor entries be directly updated in response to branch instruction execution.

Indirect branch target cache **18B** is used for indirect branch instructions. While branch instructions which form a target address from the branch displacement have static branch target addresses (at least at the virtual stage, although page mappings to physical addresses may be changed), indirect branch instructions have variable target addresses based on register and/or memory operands. Indirect branch target cache **18B** caches previously generated indirect branch target addresses in a table indexed by branch instruction address. Similar to branch predictor **18A**, indirect branch target cache **18B** is updated with actually generated

(during execution) indirect branch target addresses upon the retirement of indirect branch target instructions. In one particular embodiment, indirect branch target cache **18B** may comprise a branch target buffer having 128 entries, indexed by the least significant bits of the indirect branch instruction address, a second table having 512 entries indexed by the exclusive-OR of the least significant bits of the indirect branch instruction address (bits inverted) and least significant bits of the four indirect branch target addresses most recently predicted using the second table. The branch target buffer output is used until it mispredicts, then the second table is used until it mispredicts, etc. This structure may predict indirect branch target addresses which do not change during execution using the branch target buffer, while using the second table to predict addresses which do change during execution.

Fetch PC generation unit **18D** receives the predicted indirect branch target address from indirect branch target cache **18B**, and compares the indirect branch target address to the next fetch address generated by line predictor **12**. If the addresses do not match (and the corresponding line predictor entry is terminated by an indirect branch instruction), fetch PC generation unit **18D** signals line predictor **12** (via the status lines) that a mismatched indirect branch target has been detected. Additionally, the predicted indirect target address from indirect branch target cache **18B** is generated as the fetch address by fetch PC generation unit **18D**. Line predictor **12** compares the fetch address to fetch addresses stored therein to detect a hit and select a line predictor entry. I-cache **14** (through ITLB **60**) fetches the instruction bytes corresponding to the fetch address. It is noted that, in one embodiment, indirect branch target cache **18B** stores linear addresses and the next fetch address generated by line predictor **12** is a physical address. However, indirect branch instructions may be unconditional in such an embodiment, and the next alternate fetch address field (which is not needed to store an alternate fetch address since the branch is unconditional) may be used to store the linear address corresponding to the next fetch address for comparison purposes.

Return stack **18C** is used to predict target addresses for return instructions. As call instructions are fetched, the sequential address to the call instruction is pushed onto the return stack as a return address. As return instructions are fetched, the most recent return address is popped from the return stack and is used as the return address for that return instruction. Accordingly, if a line predictor entry is terminated by a return instruction, fetch PC generation unit **18D** compares the next fetch address from the line predictor entry to the return address provided by return address stack **18C**. Similar to the indirect target cache discussion above, if the return address and the next fetch address mismatch, fetch PC generation unit **18D** signals line predictor **12** (via the status lines) and generates the return address as the fetch address. The fetch address is searched in line predictor **12** (and translated by ITLB **60** for fetching in I-cache **14**).

The above described mechanism may allow for rapid generation of fetch addresses using line predictor **12**, with parallel verification of the predicted instruction stream using the branch predictors **18A–18C**. If the branch predictors **18A–18C** and line predictor **12** agree, then rapid instruction fetching continues. If disagreement is detected, fetch PC generation unit **18D** and line predictor **12** may update the affected line predictor entries locally.

On the other hand, certain conditions may not be detected and/or corrected by fetch PC generation unit **18D**. Predictor miss decode unit **26** may detect and handle these cases. More particularly, predictor miss decode unit **26** may decode instruction bytes when a miss is detected in line predictor **12** for a fetch address generated by fetch PC generation unit **18D**, when the next line predictor entry indication within a line predictor is invalid, or when the decode units **24A–24D** detect that instruction information from the line predictor entry is not valid. For the next line predictor indication being invalid, predictor miss decode unit **26** may provide the next fetch address as a search address to line predictor **12**. If the next fetch address hits, an indication of the corresponding line predictor entry may be recorded as the next line predictor entry indication. Otherwise, predictor miss decode unit **26** decodes the corresponding instruction bytes (received from alignment unit **12**) and generates a line predictor entry for the instructions. Predictor miss decode unit **26** communicates with fetch PC generation unit **18D** (via the line predictor update bus shown in FIG. **4**) during the generation of line predictor entries.

More particularly, predictor miss decode unit **26** may be configured to access the branch predictors **18A–18C** when terminating a line predictor entry with a branch instruction. In the present embodiment, predictor miss decode unit **26** may provide the address of the branch instruction to fetch PC generation unit **18D**, which may provide the address as the fetch PC but cancel access to line predictor **12** and ITLB **60**. In this manner, the address of the branch instruction may be provided through adder **62** (with a branch offset of zero) to branch predictor **18A** and indirect branch target cache **18B**). Alternatively, predictor miss decode unit **26** may directly access branch predictors **18A–18D** rather than providing the branch instruction address to fetch PC generation unit **18D**. The corresponding prediction information may be received by predictor miss decode unit **26** to generate next fetch address information for the generated line predictor entry. For example, if the line predictor entry is terminated by a conditional branch instruction, predictor miss decode unit **26** may use the branch prediction provided by branch predictor **18A** to determine whether to use the branch target address or the sequential address as the next fetch address. The next fetch address may be received from indirect branch target cache **18B** and may be used as the next fetch address if the line is terminated by an indirect branch instruction. The return address may be used (and popped from return stack **18C**) if the line is terminated by a return instruction.

Once the next fetch address is determined for a line predictor entry, predictor miss decode unit **26** may search line predictor **12** for the next fetch address. If a hit is detected, the hitting line predictor entry is recorded for the newly created line predictor entry and predictor miss decode unit **26** may update line predictor **12** with the new entry. If a miss is detected, the next entry to be replaced in line predictor **12** may be recorded in the new entry and predictor miss decode unit **26** may update line predictor **12**. In the case of a miss, predictor miss decode unit **26** may continue to decode instructions and generate line predictor entries until a hit in line predictor **12** is detected. In one embodiment, line predictor **12** may employ a first-in, first-out replacement policy for line predictor entries, although any suitable replacement scheme may be used.

It is noted that, in one embodiment, I-cache **14** may provide a fixed number of instruction bytes per instruction fetch, beginning with the instruction byte located by the fetch address. Since a fetch address may locate a byte anywhere within a cache line, I-cache **14** may access two cache lines in response to the fetch address (the cache line indexed by the fetch address, and a cache line at the next index in the cache). Other embodiments may limit the

number of instruction bytes provided to up to a fixed number or the end of the cache line, whichever comes first. In one embodiment, the fixed number is 16 although other embodiments may use a fixed number greater or less than 16. Furthermore, in one embodiment, I-cache **14** is set-associative. Set-associative caches provide a number of possible storage locations for a cache line identified by a particular address. Each possible storage location is a "way" of the set-associative cache. For example, in one embodiment, I-cache **14** may be 4 way set-associative and hence a particular cache line may be stored in one of 4 possible storage locations. Set-associative caches thus use two input values (an index derived from the fetch address and a way determined by comparing tags in the cache to the remaining portion of the fetch address) to provide output bytes. Rather than await the completion of tag comparisons to determine the way, line predictor **12** may store a way prediction in the line predictor entry (provided to I-cache **14** as the way prediction shown in FIG. **4**). The predicted way may be selected as the output, and the predicted way may be subsequently verified via the tag comparisons. If the predicted way is incorrect, I-cache **14** may search the other ways for a hit. The hitting way may then be recorded in line predictor **12**. Way prediction may also allow for power savings by only activating the portion of the I-cache memory comprising the predicted way (and leaving the remaining memory corresponding to the unpredicted ways idle). For embodiments in which two cache lines are accessed to provide the fixed number of bytes, two way predictions may be provided by line predictor **12** for each fetch address.

As used herein, an "address" is a value which identifies a byte within a memory system to which processor **10** is couplable. A "fetch address" is an address used to fetch instruction bytes to be executed as instructions within processor **10**. As mentioned above, processor **10** may employ an address translation mechanism in which virtual addresses (generated in response to the operands of instructions) are translated to physical addresses (which physically identify locations in the memory system). In the x86 instruction set architecture, virtual addresses may be linear addresses generated according to a segmentation mechanism operating upon logical addresses generated from operands of the instructions. Other instruction set architectures may define the virtual address differently.

Turning now to FIG. **5**, a block diagram illustrating alignment unit **16**, decode units **24A–24D**, and predecode verification circuit **66** is shown. Alignment unit **16** is coupled to receive instruction bytes from I-cache **14** and instruction information from line predictor **12**, and is coupled to provide a potential instruction (Inst. in FIG. **5**) and instruction information (Inst. Info in FIG. **5**) to each decode unit **24A–24D**. Decode units **24A–24D** are coupled to each other in series as shown in FIG. **5**, and are coupled to provide predecode invalid signals to predecode verification circuit **66**. Predecode verification circuit **66** is coupled to predictor miss decode unit **26** and to map unit **30**.

Generally, alignment unit **16** aligns a potential instruction to each of decode units **24A–24D** based on the instruction information provided by line predictor **12**. If the instruction information does not indicate an instruction for a particular decode unit **24A–24D**, that decode unit is idle for that line of instructions and does not assert its predecode invalid signal. Other decode units **24A–24D** decode the instruction provided and verify the instruction information provided to that decode unit **24A–24D**. If the instruction information is correct, then the decode unit **24A–24D** deasserts the predecode invalid signal. If the instruction information is incorrect, than the decode unit **24A–24D** asserts the predecode invalid signal.

A decode unit **24A–24D** determines that the instruction information is incorrect based on a variety of reasons. First, if the potential instruction (a portion of the instruction bytes received by alignment unit **16** which is identified as an instruction by the concurrently received instruction information) is an invalid instruction, the decode unit **24A–24D** determines that the instruction information is incorrect.

Additionally, even if the potential instruction is valid, the potential instruction may not succeed the preceding instruction within the instruction bytes (i.e. there may be one or more bytes between the end of the preceding instruction and the beginning of the instruction decoded by the decode unit **24A–24D**). In order to detect this situation, decode units **24A–24D** provide a cumulative length of the valid instructions decoded within the instruction bytes. More particularly, decode unit **24A** receives the first instruction in program order within the instruction bytes; decode unit **24B** receives the second instruction in program order within the instruction bytes; decode unit **24C** receives the third instruction in program order within the instruction bytes; and decode unit **24D** receives the fourth instruction in program order within the instruction bytes. In one embodiment, the predecode information includes an offset from the first instruction byte to the second, third, and fourth instructions. Accordingly, decode unit **24A** provides the length of the decoded instruction to decode unit **24B**, which compares the length to the provided offset. The offset should equal the length if the instruction decoded by decode unit **24B** succeeds the instruction decoded by decode unit **24B**. Decode unit **24B**, if it decodes a valid instruction, adds the length of the decoded instruction to the length provided by decode unit **24A** to generate a cumulative length of the two instructions. Decode unit **24B** provides the cumulative length to decode unit **24C**, which similarly verifies that the instruction received by decode unit **24C** succeeds the instruction decoded by decode unit **24B** and generates another cumulative length, etc.

The check to ensure that each instruction succeeds the preceding instruction detects the situation in which the instruction information does not correspond to the instruction bytes, but the byte pointed to by each instruction pointer (and possibly one or more subsequent bytes) decodes as a valid instruction. It is noted that, as an alternative to propagating cumulative lengths as shown in FIG. **5**, alignment unit **16** may calculate an expected length for each instruction from the instruction pointers and each decode unit **24A–24D** may verify that the decoded instruction is of the expected length.

Furthermore, decode units **24A–24D** may receive additional attributes included in the instruction information and may verify that the additional attributes are correct. An example of instruction information, including additional attributes, is shown below.

Thus, each decode unit **24A–24D** receives the instruction pointer used to align the potential instruction to that decode unit and any additional attributes that the decode unit may verify. Certain additional attributes may apply only to the terminating instruction within the line of instructions. In one embodiment, the terminating instruction is routed to decode unit **24D**. Thus, decode unit **24D** may receive the additional attributes which apply only to the terminating instruction and other decode units **24A–24C** may not receive this information. In other embodiments, the terminating instruction may be routed to the decode unit **24A–24D** which

would receive the instruction (based on the number of instructions within the line) and thus the additional attributes which apply to the terminating instruction may be provided to each decode unit **24A–24D** and the decode unit **24A–24D** which actually receives the terminating instruction may verify the additional attributes.

Predecode verification circuit **66** receives the predecode invalid signals from decode units **24A–24D**, and, if one or more of the predecode invalid signals is asserted, predecode verification circuit **66** may signal predictor miss decode unit **26** to predecode the corresponding instruction bytes. In one embodiment, decode units **24A–24D** complete verification of the instruction information subsequent to completing the decode of the instructions (e.g., when the corresponding decoded instruction operations are in the M1 stage of the pipeline shown in FIG. **2**). Thus, predecode verification circuit **66** may not only signal predictor miss decode unit **26** to predecode instruction bytes in response to incorrect instruction information, but may also signal one or more other units within processor **10** to discard the corresponding instruction operations as invalid. For example, map unit **30** may be signalled by predecode verification circuit **66**.

It is noted that, while predecode verification circuit **66** is shown as a separate unit in FIG. **5** for clarity in the drawing, the predecode verification circuit **66** may be integrated into one of the decode units **24A–24D**, predictor miss decode unit **26**, or any other suitable unit. Similarly, predecode verification circuit **410** may be integrated into one of decode units **408A–408D**, predecoder **412**, etc.

Turning next to FIGS. **6–9**, exemplary instruction information which may be stored for a line of instructions is shown. Other embodiments are possible and contemplated. The information shown may be employed in an embodiment of line predictor **12** employing two tables of information: a PC CAM and an index table. The PC CAM is a content addressable memory searched by the fetch address provided by fetch PC generation unit **18D**. The PC CAM is searched by predictor miss decode unit **26** when predecode instruction bytes, and is searched using the fetch address if a trap is signalled or if an incorrect next fetch address is provided by the index table. The index table stores the line predictor entries. Once a hit in the PC CAM is detected and a line predictor entry in the index table is identified, the next line predictor entry may be identified by the current line predictor entry and the PC CAM may be idle.

Turning now to FIG. **6**, a diagram illustrating an exemplary entry **90** for the PC CAM of line predictor **12** is shown. Other embodiments of the PC CAM may employ entries **90** including more information, less information, or substitute information to the information shown in the embodiment of FIG. **6**. In the embodiment of FIG. **6**, entry **90** includes a fetch address field **92**, a line predictor index field **94**, a first way prediction field **96**, and a second way prediction field **98**.

Fetch address field **92** stores the fetch address locating the first byte for which the information in the corresponding line predictor entry is stored. The fetch address stored in fetch address field **92** may be a virtual address for comparison to fetch addresses generated by fetch PC generation unit **18D**. For example, in embodiments of processor **10** employing the x86 instruction set architecture, the virtual address may be a linear address. In one embodiment, a least significant portion of the fetch address may be stored in fetch address field **92** and may be compared to fetch addresses generated by fetch PC generation unit **18D**. For example, in one particular embodiment, the least significant 18 to 20 bits may be stored and compared.

A corresponding line predictor entry within index table **72** is identified by the index stored in line predictor index field **94**. Furthermore, way predictions corresponding to the fetch address and the address of the next sequential cache line are stored in way prediction fields **96** and **98**, respectively.

Turning next to FIG. **7**, an exemplary line predictor entry **82** which may be stored in the index table of line predictor **12** is shown. Other embodiments of index table **72** may employ entries **82** including more information, less information, or substitute information to the information shown in the embodiment of FIG. **7**. In the embodiment of FIG. **7**, line predictor entry **82** includes a next entry field **100**, a plurality of instruction pointer fields **102–108**, and a control field **110**.

Next entry field **100** stores information identifying the next line predictor entry to be fetched, as well as the next fetch address. One embodiment of next entry field **100** is shown below (FIG. **8**). Control field **110** stores control information regarding the line of instructions, including instruction termination information and any other information which may be used with the line of instructions. One embodiment of control field **110** is illustrated in FIG. **9** below. Next entry field **100** and control field **110** may comprise additional attributes for this embodiment.

Each of instruction pointer fields **102–108** stores an instruction pointer for a corresponding decode unit **24A–24D**. Accordingly, the number of instruction pointer fields **102–108** may be the same as the number of decode units provided within various embodiments of processor **10**. Viewed in another way, the number of instruction pointers stored in a line predictor entry may be the maximum number of instructions which may be concurrently decoded (and processed to the schedule stage) by processor **10**. Each instruction pointer field **102–108** directly locates an instruction within the instruction bytes (as opposed to predecode data, which is stored on a byte basis and must be scanned as a whole before any instructions can be located). In one embodiment, the instruction pointers may be the length of each instruction (which, when added to the address of the instruction, locates the next instruction). A length of zero may indicate that the next instruction is invalid. Alternatively, the instruction pointers may comprise offsets from the fetch address (and a valid bit to indicate validity of the pointer). In one specific embodiment, instruction pointer **102** (which locates the first instruction within the instruction bytes) may comprise a length of the instruction, and the remaining instruction pointers may comprise offsets and valid bits. Each decode unit **24A–24D** verifies the instruction pointer of the corresponding instruction, as described above.

In one embodiment, microcode unit **28** is coupled only to decode unit **24D** (which corresponds to instruction pointer field **108**). In such an embodiment, if a line predictor entry includes a microcode (or MROM) instruction, the MROM instruction is located by instruction pointer field **108**. If the line of instructions includes fewer than the maximum number of instructions, the MROM instruction is located by instruction pointer field **108** and one or more of the instruction pointer fields **102–106** are invalid. Alternatively, the MROM instruction may be located by the appropriate instruction pointer field **102–108** based on the number of instructions in the line, and the type field **120** (shown below) may indicate that the last instruction is an MROM instruction and thus is to be aligned to decode unit **24D**.

Turning now to FIG. **8**, an exemplary next entry field **100** is shown. Other embodiments of next entry field **100** may employ more information, less information, or substitute

information to the information shown in the embodiment of FIG. **8**. In the embodiment of FIG. **8**, next entry field **100** comprises a next fetch address field **112**, a next alternate fetch address field **114**, a next index field **116**, and a next alternate index field **118**.

Next fetch address field **112** stores the next fetch address for the line predictor entry. The next fetch address is the address of the next instructions to be fetched after the line of instructions in the current entry, according to the branch prediction stored in the line predictor entry. For lines not terminated with a branch instruction, the next fetch address may be the sequential address to the terminating instruction. The next index field **116** stores the index within index table **72** of the line predictor entry corresponding to the next fetch address (i.e. the line predictor entry storing instruction pointers for the instructions fetched in response to the next fetch address). The next fetch address is verified by fetch PC generation unit **18D** and ITLB **60**, in combination, as described above.

Next alternate fetch address field **114** (and the corresponding next alternate index field **118**) are used for lines which are terminated by branch instructions (particularly conditional branch instructions). The fetch address (and corresponding line predictor entry) of the non-predicted path for the branch instruction are stored in the next alternate fetch address field **114** (and the next alternate index field **118**). In this manner, if the branch predictor **18A** disagrees with the most recent prediction by line predictor **12** for a conditional branch, the alternate path may be rapidly fetched (e.g. without resorting to predictor miss decode unit **26**). Accordingly, if the branch is predicted taken, the branch target address is stored in next fetch address field **112** and the sequential address is stored in next alternate fetch address field **114**. On the other hand, if the branch is predicted not taken, the sequential address is stored in next fetch address field **112** and the branch target address is stored in next alternate fetch address field **114**. Corresponding next indexes are stored as well in fields **116** and **118**.

In one embodiment, next fetch address field **112** and next alternate fetch address field **114** store physical addresses for addressing I-cache **14**. In this manner, the time used to perform a virtual to physical address translation may be avoided as lines of instructions are fetched from line predictor **12**. Other embodiments may employ virtual addresses in these fields and perform the translations (or employ a virtually tagged cache). It is noted that, in embodiments employing a single memory within line predictor **12** (instead of the PC CAM and index table), the index fields may be eliminated since the fetch addresses are searched in the line predictor. It is noted that the next fetch address and the next alternate fetch address may be a portion of the fetch address. For example, the in-page portions of the addresses may be stored (e.g. the least significant 12 bits) and the full address may be formed by concatenating the current page to the stored portion. It is noted that the next entry information may generally be verified by the branch prediction hardware (e.g. units **18A–18D** in FIG. **4**).

Turning next to FIG. **9**, an exemplary control field **110** is shown. Other embodiments of control field **110** may employ more information, less information, or substitute information to the information shown in the embodiment of FIG. **9**. In the embodiment of FIG. **9**, control field **110** includes a last instruction type field **120**, a branch prediction field **122**, a branch displacement field **124**, a continuation field **126**, a line/bank cross field **127**, a first way prediction field **128**, a second way prediction field **130**, and an entry point field **132**.

Last instruction type field **120** stores an indication of the type of the last instruction (or terminating instruction) within the line of instructions. The type of instruction may be provided to fetch PC generation unit **18D** to allow fetch PC generation unit **18D** to determine which of branch predictors **18A–18C** to use to verify the branch prediction within the line predictor entry. More particularly, last instruction type field **120** may include encodings indicating sequential fetch (no branch), microcode instruction, conditional branch instruction, indirect branch instruction, call instruction, and return instruction. The conditional branch instruction encoding results in branch predictor **18A** being used to verify the direction of the branch prediction. The indirect branch instruction encoding results in the next fetch address being verified against indirect branch target cache **18B**. The return instruction encoding results in the next fetch address being verified against return stack **18C**. The last instruction type field **120** may be verified by the decode unit **24A–24D** receiving the terminating instruction.

Branch prediction field **122** stores the branch prediction recorded by line predictor **12** for the branch instruction terminating the line (if any). Generally, fetch PC generation unit **18D** verifies that the branch prediction in field **122** matches (in terms of taken/not taken) the prediction from branch predictor **18A**. In one embodiment, branch prediction field **122** may comprise a bit with one binary state of the bit indicating taken (e.g. binary one) and the other binary state indicating not taken (e.g. binary zero). If the prediction disagrees with branch predictor **122**, the prediction may be switched. In another embodiment, branch prediction field **122** may comprise a saturating counter with the binary state of the most significant bit indicating taken/not taken. If the taken/not taken prediction disagrees with the prediction from branch predictor **18A**, the saturating counter is adjusted by one in the direction of the prediction from branch predictor **18A** (e.g. incremented if taken, decremented if not taken). The saturating counter embodiment may more accurately predict loop instructions, for example, in which each N-1 taken iterations (where N is the loop count) is followed by one not taken iteration.

Branch displacement field **124** stores an indication of the branch displacement corresponding to a direct branch instruction. In one embodiment, branch displacement field **124** may comprise an offset from the fetch address to the first byte of the branch displacement. Fetch PC generation unit **18D** may use the offset to locate the branch displacement within the fetched instruction bytes, and hence may be used to select the displacement from the fetched instruction bytes. In another embodiment, the branch displacement may be stored in branch displacement field **124**, which may be directly used to determine the branch target address. Branch displacement field **124** may be verified by the decode unit **24A–24D** receiving the branch instruction.

In the present embodiment, the instruction bytes represented by a line predictor entry may be fetched from two consecutive cache lines of instruction bytes. Accordingly, one or more bytes may be in a different page than the other instruction bytes. Continuation field **126** is used to signal the page crossing, so that the fetch address corresponding to the second cache line may be generated and translated. Once a new page mapping is available, other fetches within the page have the correct physical address as well. The instruction bytes in the second page are then fetched and merged with the instruction bytes within the first page. Continuation field **126** may comprise a bit indicative, in one binary state, that the line of instructions crosses a page boundary, and indicative, in the other binary state, that the line of instruc-

tions does not cross a page boundary. Continuation field **126** may also be used to signal a branch target address which is in a different page than the branch instruction.

To conserve power, I-cache **14** may be configured to access only the tag for the first cache line if the fetched instruction bytes do not cross a cache line boundary. Additionally, I-cache **14** may organize the instruction bytes memory as a set of banks, and may access only the bank including the instruction bytes if instruction bytes are included within one bank. The other tags/banks may be idle, and thus not consume power. Line/Bank cross field **127** may store indications of whether the instruction bytes cross a cache line boundary and/or a bank boundary. The line and bank indications may be verified by I-cache **14** and may be updated directly into line predictor **12** if incorrect. Alternatively, the line and bank indications may be verified by decode units **24A–24D**.

Similar to way prediction fields **96** and **98**, way prediction fields **128** and **130** store the way predictions corresponding to the next fetch address (and the sequential address to the next fetch address). Way predictions may be verified by I-cache **14**, and updated directly into entry **90** or **82**. Finally, entry point field **132** may store an entry point for a microcode instruction within the line of instructions (if any). An entry point for microcode instructions is the first address within the microcode ROM at which the microcode routine corresponding to the microcode instruction is stored. If the line of instructions includes a microcode instruction, entry point field **132** stores the entry point for the instruction. The time used to decode the microcode instruction to determine the entry point may be eliminated during the fetch and dispatch of the instruction, allowing for the microcode routine to be entered more rapidly. The stored entry point may be verified against an entry point generated in response to the instruction (by decode unit **24D** or MROM unit **28**). It is noted that, in one embodiment in which both branch instructions and microcode instructions are terminating conditions for a line, branch displacement field **124** and entry point field **132** may be overlapped.

Turning next to FIG. **10**, a flowchart is shown illustrating the operation of decode units **24A–24D** for validating instruction information as illustrated in FIGS. **6–9**. Other embodiments are possible and contemplated. Although illustrated in a serial order for ease of understanding, the steps shown in FIG. **10** may be performed in any suitable order and/or may be performed in parallel by the combinatorial logic within the decode units. Particularly, decision blocks **140, 142, 144, 146,** and **148** may be performed in parallel or in any desired order.

The decode unit decodes the received instruction bytes and determines if the instruction is valid (decision block **140**). If the instruction is invalid, the decode unit signals that the predecode data is invalid (step **152**). The decode unit also determines if the instruction pointer is valid (e.g. that the instruction pointer points to an instruction byte which succeeds the preceding instruction in program order—decision block **142**). If the instruction pointer is invalid, the decode unit signals that the predecode data is invalid (step **152**). If the instruction received by the decode unit is the terminating instruction within the line of instructions, the decode unit verifies that the instruction type from instruction type field **120** matches the instruction type of the decoded instruction (decision block **144**). If the instruction type does not match, the decode unit signals that the predecode data is invalid (step **152**). The decode unit determines if the instruction is a microcode instruction (decision block **146**), and generates an entry point for the instruction if the instruction is a microcode instruction. The generated entry point is compared to the entry point from the line predictor entry (decision block **154**), and if the entry points do not match, the decode unit signals that the predecode data is invalid (step **152**). The decode unit also determines if the instruction is a branch instruction (decision block **148**), and verifies the branch displacement indication from the line predictor entry if the instruction is a branch instruction (decision block **156**). If the branch displacement indication is invalid, then the decode unit signals that the predecode data is invalid (step **152**). If none of the verification checks results in a signalling that the predecode data is invalid, the decode unit signals that the predecode data is valid (step **150**).

It is noted that, in embodiments in which the terminating instruction is routed to decode unit **24D** only, decode units **24A–24C** may be configured to perform only decision blocks **140** and **142** and step **152**, while decode unit **24D** may perform each of the checks shown in the flowchart.

Turning next to FIG. **11**, a block diagram of an exemplary decode unit **160** is shown. Other embodiments are possible and contemplated, including Boolean equivalents of the combinatorial logic circuits shown in FIG. **11**. In the embodiment of FIG. **11**, decode unit **160** includes a comparator **162**, an adder **164**, a decoder **166**, an entry point generator **168**, an OR gate **170**, and an inverter **172**. Comparator **162** is coupled to receive an instruction pointer from alignment unit **16** and a cumulative length from a preceding decode unit, and is further coupled through inverter **172** to OR gate **170**. Adder circuit **164** is coupled to receive the cumulative length and a length from decoder **166**, and is coupled to provide a cumulative length to a succeeding decode unit. Decoder **166** is coupled to receive an instruction type, a branch displacement indication, and an instruction from alignment unit **16**, and is further coupled to OR gate **170** and to provide instruction operations to map unit **30**. Entry point generator **168** is coupled to receive the instruction and an entry point from alignment unit **16** and is coupled to OR gate **170**.

Comparator **162** compares the instruction pointer used by alignment unit **16** to align the instruction to decode unit **160** to the cumulative length received from the preceding decode unit (i.e. the decode unit which decodes the instruction preceding the instruction decoded by decode unit **160**). If the cumulative length matches the instruction pointer, then the instruction pointer is verified as correct. If a mismatch is detected, the instruction pointer is incorrect and the predecode invalid signal is to be asserted. Since comparator **162** asserts its output in response to detecting a match, inverter **172** is provided to invert the output of comparator **162** to provide an input to OR gate **170**. Thus, a mismatch detected by comparator **162** results in an assertion of the predecode invalid signal (which is the output of OR gate **170**). Since decode unit **24A** receives the first instruction, there is no cumulative length to compare against. Accordingly, decode unit **24A** may omit comparator **162**. Other decode units **24B–24D** may include circuitry similar to comparator **162**.

Adder **164** adds the cumulative length received from the preceding decode unit to the length of the instruction decoded by decoder **166** to produce a new cumulative length for the succeeding decode unit (i.e. the decode unit which decodes the instruction succeeding the instruction decoded by decode unit **160**). Since decode unit **24D** in the above embodiment does not have a succeeding decode unit, decode unit **24D** may omit adder **164**. Additionally, since decode unit **24A** does not receive a cumulative length, decode unit **24A** may omit adder **164** and provide the length detected by decoder **166** as the cumulative length. Decode units **24B–24C** may include circuitry similar to adder **164**.

Decoder **166** decodes the received instruction and provides instruction operations to map unit **30** in response to the received instruction. If the received instruction is invalid, decoder **166** asserts an output signal to OR gate **170**, causing assertion of the predecode invalid signal. Additionally, decoder **166** verifies the last instruction type and branch displacement additional attributes from the predecode data (if decoder **166** decodes the terminating instruction within the line). If either the last instruction type or the branch displacement indication are incorrect, decoder **166** asserts the output signal to OR gate **170**. It is noted that, in embodiments in which decode unit **24D** receives the terminating instruction, the decoder circuitry within decode units **24A–24C** may decode the received instruction but may not receive instruction type or branch displacement information. In embodiments in which the terminating instruction may be provided to any decode unit **24A–24D**, decode units **24A–24D** may include decoder circuitry similar to decoder **166**.

Entry point generator **168** decodes the received instruction to generate an entry point if the received instruction is a microcode instruction. Entry point generator **168** receives the entry point recorded in the line predictor entry, and compares that entry point to the generated entry point. If the entry points do not match, entry point generator **168** asserts an output signal to OR gate **170**, causing an assertion of the predecode invalid signal. It is noted that entry point generator **168** may also provide the generated entry point to predictor miss decode unit **26** for inclusion in the corrected line predictor entry. In the present embodiment, decode unit **24D** may include circuitry similar to entry point generator **168** (since decode unit **24D** is coupled to microcode unit **28**).

Turning next to FIG. **12**, a timing diagram is shown illustrating the detection of invalid predecode data by decode units **24A–24D**. The timing diagram illustrates a set of clock cycles delimited by vertical dashed lines, with a label for the clock cycle above and between (horizontally) the vertical dashed lines for that clock cycle. Each clock cycle will be referred to with the corresponding label. The pipeline stage labels shown in FIG. **2** are used in the timing diagram. This case may occur due to address aliasing, for example, or due to the lack of coherency maintained between the line predictor and the I-cache.

The instruction bytes and instruction information are fetched in the line predictor stage (clock cycle CLK**1**) and flow through the instruction cache and alignment stages (clock cycles CLK**2** and CLK**3**). Alignment unit **16** uses the provided instruction information to align instructions to decode units **24A–24D**. The decode units **24A–24D** decode the provided instructions (Decode stage, clock cycle CLK**4**). Additionally, the decode units **24A–24D** signal with an indication of whether or not the predecode data is invalid. In the present embodiment, the determination of valid or invalid predecode occurs after the decode stage is complete (e.g. clock cycle CLK**5** in FIG. **12**), the instruction bytes are routed to predictor miss decode unit **26**, which begins predecoding the instruction bytes (clock cycle CLK**6**). It is noted that predictor miss decode unit **26** may speculatively begin decoding at clock cycle CLK**4**, if desired. It is further noted that the first instruction in the instruction bytes (decoded by decode unit **24A**) is valid and may be dispatched even as predictor miss decode unit **26** decodes the remaining instructions.

Computer Systems

Turning now to FIG. **13**, a block diagram of one embodiment of a computer system **200** including processor **10** coupled to a variety of system components through a bus bridge **202** is shown. Other embodiments are possible and contemplated. In the depicted system, a main memory **204** is coupled to bus bridge **202** through a memory bus **206**, and a graphics controller **208** is coupled to bus bridge **202** through an AGP bus **210**. Finally, a plurality of PCI devices **212A–212B** are coupled to bus bridge **202** through a PCI bus **214**. A secondary bus bridge **216** may further be provided to accommodate an electrical interface to one or more EISA or ISA devices **218** through an EISA/ISA bus **220**. Processor **10** is coupled to bus bridge **202** through a CPU bus **224** and to an optional L2 cache **228**. Together, CPU bus **224** and the interface to L2 cache **228** may comprise external interface **52**.

Bus bridge **202** provides an interface between processor **10**, main memory **204**, graphics controller **208**, and devices attached to PCI bus **214**. When an operation is received from one of the devices connected to bus bridge **202**, bus bridge **202** identifies the target of the operation (e.g. a particular device or, in the case of PCI bus **214**, that the target is on PCI bus **214**). Bus bridge **202** routes the operation to the targeted device. Bus bridge **202** generally translates an operation from the protocol used by the source device or bus to the protocol used by the target device or bus.

In addition to providing an interface to an ISA/EISA bus for PCI bus **214**, secondary bus bridge **216** may further incorporate additional functionality, as desired. An input/output controller (not shown), either external from or integrated with secondary bus bridge **216**, may also be included within computer system **200** to provide operational support for a keyboard and mouse **222** and for various serial and parallel ports, as desired. An external cache unit (not shown) may further be coupled to CPU bus **224** between processor **10** and bus bridge **202** in other embodiments. Alternatively, the external cache may be coupled to bus bridge **202** and cache control logic for the external cache may be integrated into bus bridge **202**. L2 cache **228** is further shown in a backside configuration to processor **10**. It is noted that L2 cache **228** may be separate from processor **10**, integrated into a cartridge (e.g. slot **1** or slot A) with processor **10**, or even integrated onto a semiconductor substrate with processor **10**.

Main memory **204** is a memory in which application programs are stored and from which processor **10** primarily executes. A suitable main memory **204** comprises DRAM (Dynamic Random Access Memory). For example, a plurality of banks of SDRAM (Synchronous DRAM) or Rambus DRAM (RDRAM) may be suitable.

PCI devices **212A–212B** are illustrative of a variety of peripheral devices such as, for example, network interface cards, video accelerators, audio cards, hard or floppy disk drives or drive controllers, SCSI (Small Computer Systems Interface) adapters and telephony cards. Similarly, ISA device **218** is illustrative of various types of peripheral devices, such as a modem, a sound card, and a variety of data acquisition cards such as GPIB or field bus interface cards.

Graphics controller **208** is provided to control the rendering of text and images on a display **226**. Graphics controller **208** may embody a typical graphics accelerator generally known in the art to render three-dimensional data structures which can be effectively shifted into and from main memory **204**. Graphics controller **208** may therefore be a master of AGP bus **210** in that it can request and receive access to a target interface within bus bridge **202** to thereby obtain access to main memory **204**. A dedicated graphics bus accommodates rapid retrieval of data from main memory **204**. For certain operations, graphics controller **208** may further be configured to generate PCI protocol transactions

on AGP bus **210**. The AGP interface of bus bridge **202** may thus include functionality to support both AGP protocol transactions as well as PCI protocol target and initiator transactions. Display **226** is any electronic display upon which an image or text can be presented. A suitable display **226** includes a cathode ray tube ("CRT"), a liquid crystal display ("LCD"), etc.

It is noted that, while the AGP, PCI, and ISA or EISA buses have been used as examples in the above description, any bus architectures may be substituted as desired. It is further noted that computer system **200** may be a multiprocessing computer system including additional processors (e.g. processor **10***a* shown as an optional component of computer system **200**). Processor **10***a* may be similar to processor **10**. More particularly, processor **10***a* may be an identical copy of processor **10**. Processor **10***a* may be connected to bus bridge **202** via an independent bus (as shown in FIG. **13**) or may share CPU bus **224** with processor **10**. Furthermore, processor **10***a* may be coupled to an optional L2 cache **228***a* similar to L2 cache **228**.

Turning now to FIG. **14**, another embodiment of a computer system **300** is shown. Other embodiments are possible and contemplated. In the embodiment of FIG. **14**, computer system **300** includes several processing nodes **312A**, **312B**, **312C**, and **312D**. Each processing node is coupled to a respective memory **314A–314D** via a memory controller **316A–316D** included within each respective processing node **312A–312D**. Additionally, processing nodes **312A–312D** include interface logic used to communicate between the processing nodes **312A–312D**. For example, processing node **312A** includes interface logic **318A** for communicating with processing node **312B**, interface logic **318B** for communicating with processing node **312C**, and a third interface logic **318C** for communicating with yet another processing node (not shown). Similarly, processing node **312B** includes interface logic **318D**, **318E**, and **318F**; processing node **312C** includes interface logic **318G**, **318H**, and **318I**; and processing node **312D** includes interface logic **318J**, **318K**, and **318L**. Processing node **312D** is coupled to communicate with a plurality of input/output devices (e.g. devices **320A–320B** in a daisy chain configuration) via interface logic **318L**. Other processing nodes may communicate with other I/O devices in a similar fashion.

Processing nodes **312A–312D** implement a packet-based link for inter-processing node communication. In the present embodiment, the link is implemented as sets of unidirectional lines (e.g. lines **324A** are used to transmit packets from processing node **312A** to processing node **312B** and lines **324B** are used to transmit packets from processing node **312B** to processing node **312A**). Other sets of lines **324C–324H** are used to transmit packets between other processing nodes as illustrated in FIG. **14**. Generally, each set of lines **324** may include one or more data lines, one or more clock lines corresponding to the data lines, and one or more control lines indicating the type of packet being conveyed. The link may be operated in a cache coherent fashion for communication between processing nodes or in a noncoherent fashion for communication between a processing node and an I/O device (or a bus bridge to an I/O bus of conventional construction such as the PCI bus or ISA bus). Furthermore, the link may be operated in a noncoherent fashion using a daisy-chain structure between I/O devices as shown. It is noted that a packet to be transmitted from one processing node to another may pass through one or more intermediate nodes. For example, a packet transmitted by processing node **312A** to processing node **312D** may pass through either processing node **312B** or processing node **312C** as shown in FIG. **14**. Any suitable routing algorithm may be used. Other embodiments of computer system **300** may include more or fewer processing nodes then the embodiment shown in FIG. **14**.

Generally, the packets may be transmitted as one or more bit times on the lines **324** between nodes. A bit time may be the rising or falling edge of the clock signal on the corresponding clock lines. The packets may include command packets for initiating transactions, probe packets for maintaining cache coherency, and response packets from responding to probes and commands.

Processing nodes **312A–312D**, in addition to a memory controller and interface logic, may include one or more processors. Broadly speaking, a processing node comprises at least one processor and may optionally include a memory controller for communicating with a memory and other logic as desired. More particularly, a processing node **312A–312D** may comprise processor **10**. External interface unit **46** may includes the interface logic **318** within the node, as well as the memory controller **316**.

Memories **314A–314D** may comprise any suitable memory devices. For example, a memory **314A–314D** may comprise one or more RAMBUS DRAMs (RDRAMs), synchronous DRAMs (SDRAMs), static RAM, etc. The address space of computer system **300** is divided among memories **314A–314D**. Each processing node **312A–312D** may include a memory map used to determine which addresses are mapped to which memories **314A–314D**, and hence to which processing node **312A–312D** a memory request for a particular address should be routed. In one embodiment, the coherency point for an address within computer system **300** is the memory controller **316A–316D** coupled to the memory storing bytes corresponding to the address. In other words, the memory controller **316A–316D** is responsible for ensuring that each memory access to the corresponding memory **314A–314D** occurs in a cache coherent fashion. Memory controllers **316A–316D** may comprise control circuitry for interfacing to memories **314A–314D**. Additionally, memory controllers **316A–316D** may include request queues for queuing memory requests.

Generally, interface logic **318A–318L** may comprise a variety of buffers for receiving packets from the link and for buffering packets to be transmitted upon the link. Computer system **300** may employ any suitable flow control mechanism for transmitting packets. For example, in one embodiment, each interface logic **318** stores a count of the number of each type of buffer within the receiver at the other end of the link to which that interface logic is connected. The interface logic does not transmit a packet unless the receiving interface logic has a free buffer to store the packet. As a receiving buffer is freed by routing a packet onward, the receiving interface logic transmits a message to the sending interface logic to indicate that the buffer has been freed. Such a mechanism may be referred to as a "coupon-based" system.

I/O devices **320A–320B** may be any suitable I/O devices. For example, I/O devices **320A–320B** may include network interface cards, video accelerators, audio cards, hard or floppy disk drives or drive controllers, SCSI (Small Computer Systems Interface) adapters and telephony cards, modems, sound cards, and a variety of data acquisition cards such as GPIB or field bus interface cards.

Numerous variations and modifications will become apparent to those skilled in the art once the above disclosure is fully appreciated. It is intended that the following claims be interpreted to embrace all such variations and modifications.

What is claimed is:

1. A processor comprising:

a predecode cache configured to store predecode information, wherein said predecode cache is configured to output said predecode information responsive to a fetch address; and

one or more decode units coupled to receive said predecode information, wherein each decode unit is coupled to receive a portion of a plurality of instruction bytes fetched in response to said fetch address, and wherein said each decode unit is configured to decode said portion, and wherein said one or more decode units are configured to verify that said predecode information corresponds to said plurality of instruction bytes.

2. The processor as recited in claim 1 further comprising a predecode verification circuit coupled to said one or more decode units, wherein said each decode unit is configured to signal a result of verifying said predecode information to said predecode verification circuit, and wherein said predecode verification circuit is configured to initiate predecoding if at least one of said one or more decode units indicates that said predecode information does not correspond to said plurality of instruction bytes.

3. The processor as recited in claim 2 wherein said processor is configured to dispatch instructions from said plurality of instruction bytes as said instructions are predecoded.

4. The processor as recited in claim 1 further comprising:

a fetch PC generation unit configured to generate a virtual fetch address; and

a translation lookaside buffer (TLB) coupled to said fetch PC generation unit, wherein said TLB is configured to translate said virtual fetch address to a physical fetch address;

wherein said predecode information includes a physical next fetch address, and wherein said TLB is configured to verify that said next fetch address matches a corresponding physical fetch address.

5. The processor as recited in claim 1 wherein a first decode unit of said one or more decode units is configured to determine that said predecode information does not correspond to said plurality of instruction bytes if said portion is not a valid instruction.

6. The processor as recited in claim 5 further comprising an alignment unit coupled receive said predecode information and said plurality of instruction bytes, and wherein said alignment unit is configured to align said portions of said plurality of instruction bytes to said one or more decode units responsive to said predecode information, and wherein said predecode information does not correspond to said plurality of instruction bytes if said predecode information locates an instruction incorrectly within said plurality of instruction bytes.

7. The processor as recited in claim 6 wherein said predecode information comprises a plurality of instruction pointers, and wherein each of said plurality of instruction pointers points to a byte within said plurality of instruction bytes, and wherein one of said predecode information does not correspond to said plurality of instruction bytes if said byte pointed to by one of said plurality of instruction pointers does not succeed a last byte of a preceding instruction within said plurality of instruction bytes.

8. The processor as recited in claim 7 wherein first decode unit is coupled to receive an indication of said last byte of said preceding instruction from another one of said one or more decode units to determine if said byte succeeds said last byte.

9. The processor as recited in claim 1 wherein said predecode information comprises an instruction type of a last instruction identified by said predecode information, and wherein a first decode unit of said one or more decode units which receives said last instruction is configured to verify that said instruction type matches a decoded instruction type of said last instruction.

10. The processor as recited in claim 1 wherein said predecode information comprises an entry point if a microcode instruction is included in said plurality of instruction bytes, and wherein a first decode unit of said one or more decode units which receives said microcode instruction is configured to verify that said microcode instruction is included within said plurality of instruction bytes and to generate an entry point for said microcode instruction, and wherein said first decode unit is configured to verify that said entry point from said predecode information matches said entry point generated for said microcode instruction.

11. The processor as recited in claim 1 wherein said predecode information comprises a branch displacement identifier corresponding to a branch instruction within said plurality of instruction bytes, and wherein a first decode unit of said one or more decode units receiving said branch instruction is configured to verify said branch displacement identifier.

12. The processor as recited in claim 1 further comprising an instruction cache, and wherein said predecode information includes a bank crossing indication identifying whether or not said plurality of instruction bytes crosses a bank boundary within said instruction cache, and wherein said one or more decoders are configured to verify if said bank crossing indication corresponds to said plurality of instruction bytes.

13. The processor as recited in claim 1 further comprising an instruction cache, and wherein said predecode information includes a line crossing indication identifying whether or not said plurality of instruction bytes crosses a cache line boundary within said instruction cache, and wherein said one or more decoders are configured to verify if said line crossing indication corresponds to said plurality of instruction bytes.

14. A computer system comprising:

a processor including:

a predecode cache configured to store predecode information, wherein said predecode cache is configured to output said predecode information responsive to a fetch address; and

one or more decode units coupled to receive said predecode information, wherein each decode unit is coupled to receive a portion of a plurality of instruction bytes fetched in response to said fetch address, and wherein said each decode unit is configured to decode said portion, and wherein said one or more decode units are configured to verify that said predecode information corresponds to said plurality of instruction bytes; and

an input/output (I/O) device configured to communicate between said computer system and another computer system to which said I/O device is couplable.

15. The computer system as recited in claim 14 wherein said I/O device comprises a modem.

16. The computer system as recited in claim 14 further comprising an audio I/O device.

17. The computer system as recited in claim 16 wherein said audio I/O device comprises a sound card.

18. A method comprising:

fetching predecode information from a predecode cache responsive to a fetch address;

fetching a plurality of instruction bytes responsive to said fetch address;

decoding said plurality of instruction bytes; and

verifying that said predecode information corresponds to said plurality of instruction bytes.

19. The method as recited in claim **18** further comprising locating instructions within said plurality of instruction bytes using said predecode information; and wherein said verifying comprises determining that said predecode information does not correspond to said plurality of instruction bytes if one or more of said instructions located during said locating is invalid.

20. The method as recited in claim **19** wherein said predecode information comprises a plurality of instruction pointers, and wherein each of said plurality of instruction pointers points to a byte, and wherein said verifying comprises verifying that said byte succeeds a last byte of another instruction within said plurality of instruction bytes.

21. The processor as recited in claim **5** wherein the first decode unit is configured to determine that said predecode information does not correspond to said pluality of instruction bytes if said portion is said valid instruction but one or more attributes of said valid instruction identified by said predecode information do not correspond to said valid instruction.

22. The method as recited in claim **19** wherein said verifying further comprises determining that said predecode information does not correspond to said plurality of instruction bytes if the instruction located during said locating are valid but one or more attributes of said instruction identified by said predecode information do not correspond to said valid instruction.

* * * * *